Undecidability of The Halting Problem

A Formal Proof in Agda

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Zongzhe Yuan

Computer Science

University of Nottingham

A thesis submitted for the degree of

Bachelor of Science

May 2016

Acknowledgements

Some People helped with the preparation of the paper. Three in particular have made significant contributions to the content of the paper, the style of the paper, as well as the code of the proof in **Agda** : Gabe Dijkstra helped a lot with the professional **Agda** technique. Ambrus Kaposi helped significantly with many proofs and several lemmas involving some key point in the proof of the correctness of the universal model and the proof of Halting Problem. Thorsten Altenkirch, as the supervisor of the project, provided the overview and the general idea of the proof of the Halting Problem in **Agda**, and helped me a lot during the whole project. Part of the definition, including the definition of **WHILE** language used in the project follows the definition in the paper *Computability and complexity: from a programming perspective* by *Jones, Neil D* in 1997[12].

Abstract

The *Halting Problem*, is a famous problem in computability theory, proved by Alan Turing in 1936 [12]. By choosing some computational model which is *Turing-equivalents* to the *Turing machine*[10], defining the syntax and semantics of that model, constructing the universal computation model by the theory of *Turing-complete*, one could proof the undecidability of the *Halting Problem.* The aim of the project is constructing a proper computational model, proving the correctness of the universal model (the property of *Turing completeness*), and finally using that model to prove the undecidability of *halting problem* by contradiction.

Keywords: Agda, Turing-completeness, Universal Model, Undecidable

Contents

Chapter 1

Approach to the Project

1.1 Overview

Alan Turing analysed and formalised the class of all computational procedure in 1936[16] which comes the famous computation model *Turing machine*. Alan Turing used the model of *Turing machine* to answer the question "Does a machine exist that can determine whether any arbitrary machine on its tape is 'circular'", and proved the undecidability of *Halting problem* in 1936 as well[16] which is the object to this paper.

To prove the undecidability of *Halting problem*, one must use a computational model that is *Turing-equivalent* to *Turing machine* by *Church-Turing thesis*[10]. For simplicity, we choose a computational model that contains one *input*, one *output* and a *Function* from *input* to *output*. We choose some proper data structure D to represent the data and let the *input* and *output* belongs to D. Then we could define the *Function* as a relation $\bullet \to \bullet \subseteq D \times D$ which is also the mapping from the *input* to the *output*. For the purpose of proving, we must construct this computational model with concrete syntax and semantics using some proof assistant language – the meta language.

Then we should constructing the universal model following the thesis of *Turing-complete*[10]. That is, if we have $f \in \mathit{Function}$, *input* and *output* $\in D$, and $f(\mathit{input}) \equiv \mathit{output}$, then the universal model u has the property that $u(\lfloor f \rfloor \cdot \mathit{input}) \equiv \mathit{output}$ where $\lfloor f \rfloor$ is the code of *Function* f in data format D and $\cdot$ is the concat symbol in the data structure D.

We must prove the correctness of out predefined universal computation model. During that proof, we should define the concrete code method to code the *Function* into

our predefined data structure D without ambiguity. The first step in our proof is constructing the interpretation simulation step s in the chosen meta language and proving the relation between the *Function* f and the interpretation simulation s. We could define the interpretation simulation as a stack machine: $(Command, Stack, Variable) \Rightarrow (Command', Stack', Variable') \subseteq (D,D,D) \times (D,D,D)$ which is a one-step relation between two triples. Then we should prove that if we have $f \in$ *Function*, *input* and *output* $\in D$, and $f(input) \equiv output$, then we could get $(\lfloor f \rfloor, \epsilon, input) \Rightarrow^* (\epsilon, \epsilon, output)$ where $\Rightarrow^*$ is several steps relation of relation $\Rightarrow$. From the previous proof we know that we can using the meta language to simulate the one step of the *Function* in several steps. Then we can construct the universal model, and prove that the interpretation step simulated in the meta language has correspondence to the interpretation in the universal model. If we have $f \in$ *Function*, *input* and *output* $\in D$, and $(\lfloor f \rfloor, \epsilon, input) \Rightarrow^* (\epsilon, \epsilon, output)$, then the universal model u has the property that $u(\lfloor f \rfloor \cdot input) \equiv output$. The proof is inducted by the step of relation $\Rightarrow$. Finally we can conclude that the program and the universal model has the correspondence that for $f \in$ *Function*, *input* and *output* $\in D$, and $f(input) \equiv output$, then for the universal model u, $u(\lfloor f \rfloor \cdot input) \equiv output$.
To prove the undecidability of *halting problem*, we could assume that there exists a program $h \in$ Program that has some properties: $\forall p \in$ Program and $\forall input \in$ Data, if p halt on *input*, then $h(\lfloor p \rfloor \cdot input) \equiv 1$, else $h(\lfloor p \rfloor \cdot input) \equiv 0$, which means h decide the *halting problem*. Then we could construct a program $m \in$ Program and feed h to m which means run h inside m. Then we let h to decide $m(\lfloor m \rfloor)$ will halt or not, which will cause contradiction to the definition of m. Finally we could conclude that h doesn't exists by contradiction, and there is no model that could decide the *halting problem* which means the *halting problem* is undecidable.

1.2 Background

1.2.1 Church-Turing Thesis

Church-Turing thesis is a hypothesis about the nature of computable functions[8]. The thesis states that every effective computation can be carried out by a *Turing machine*[10]. Turing gave the definition of his thesis as the LCMs [logical computing machines: Turing's expression for *Turing machines*] can do anything, that could be described as "rule of thumb" or "purely mechanical" (Turing 1948:7.)[10].

The two basic concept that related to this paper are *Turing completeness* and *Truing equivalence*.

Turing completeness is the concept in computability theory, such that a computational model (for example a programming language, or recursive function) is Turing complete if and only if the model could be used to simulate any single-taped *Turing machine*[15].

Truing equivalence said if two computational model could simulate each one by the other, then these two computational machine is *Turing equivalence*[15].
By *Church-Turing Thesis*, any function that can be computed by some algorithm can be computed by a *Turing machine*[8]. Thus there are many computational models that is *Turing equivalent* to a *Truing machine*[10]. Though the goal of the project is to prove the undecidability of *halting problem*, constructing and formalise a universal *Turing machine* is quite complex. Thus we can choose many other notions of effective procedure than *Turing machine* (means that the notion is *Turing-complete*):

- Recursive functions as defined by Kleene [18]
- The lambda calculus approach to function definitions due to Church [13]
- Random access machines [9]
- Markov algorithms[3]

Considering those different formalism which have the same efficient computing module, there are several common characteristics[12].

- The procedure consist of finite size of instructions.
- The computation is carried out in a discrete stepwise fashion but not continuous methods or analogue devices.
- The computation is carried out deterministically but not random methods.
- Though a terminating computation must not rely on an infinite amount of space or time, there is no bound on the amount of memory storage space or time available.

Universal Turing Machine is a Turing machine that can simulate an arbitrary *Turing machine* on arbitrary input[6]. That is for $t \in \mathbf{TM}$, *input* and *output* $\in \Sigma^*$ (the set of input symbols on Kleene star[11]) , and $t(\mathit{input})$ yields *output*, then the universal turing machine $u \in \mathbf{UTM}$ has the property that $u(\lfloor t \rfloor \cdot \mathit{input})$ yields *output* where $\lfloor t \rfloor$ is the code of *Turing Machine t*.
Those computational model listed above have been proved to be *Turing-equivalence* to the *Turing machine*, which means they also have the property of *Turing completeness* and could construct universal computational model.

1.2.2 Decidable and Non-Decidable

In the area of computability, a set **S** is **Recursive** (**Decidable**) $\iff$ given a set **D** and $\mathbf{S} \in \mathbf{D}$, there is a function f applies to the element $a \in \mathbf{D}$, f will return "true" if $a \in \mathbf{S}$ and f will return "false" if $a \notin \mathbf{S}$[12]. **Decidable** set is closed under union, intersection, complement difference and Kleene star[4].
A set **S** is **Recursively Enumerable** (**Semi-Decidable**) $\iff$ given a set **D** and $\mathbf{S} \in \mathbf{D}$, there is a function f applies to the element $a \in \mathbf{D}$, f will return "true" if $a \in \mathbf{S}$ and f will return "false" or never terminate if $a \notin \mathbf{S}$ which means no guarantee to terminate under the element $a \notin \mathbf{S}$[12]. If a set **S** is recursively enumerable and the implement of **S** is also recursively enumerable, then set **S** is Recursive[5].

1.2.3 Halting Problem

In area of computability theory, the *Halting Problem* is the problem that a given universal computing program (the model that is *Turing-complete* [12]) could determine any other arbitrary computing program that would return the result on arbitrary input in a finite number of steps (a finite amount of time), or would run forever [2].
That is, $\forall t \in \mathbf{TM}$, $\mathit{input} \in \Sigma^*$, if there exists $h \in \mathbf{TM}$ such that, if $\mathit{halt}_t(\mathit{input})$, then $h(\lfloor t \rfloor \cdot \mathit{input}) \equiv \mathit{true}$, else $h(\lfloor t \rfloor \cdot \mathit{input}) \equiv \mathit{false}$, then h determining the *halting problem*.
It is easily to prove that *halting problem* is semi-decidable[2] because $\forall t \in \mathbf{TM}$, $\mathit{input} \in \Sigma^*$, if eventually $\mathit{halt}_t(\mathit{input})$ then we can easily get the result. However whether the halting problem is decidable or not is interesting, and to prove the undecidable of the *halting problem* is the main target of this paper.

1.2.4 WHILE language

The **WHILE** language is a language that has just the right mix of expressive power and simplicity. It has strict definition of syntax and semantics. And it has the same computing effective level of *Turing machine* model (Turing-complete)[12]. And also the data structure of **WHILE** treat the program as data object which could sole some rather complex missions. Furthermore with the simplicity, **WHILE** language could simply be used to prove many theorems and their behaviours. By considering those several reasons, the project will focus on proving the *halting problem* undecidability on the model of **WHILE** language[12].

1.2.5 Agda

Agda is a dependently type language[14] that is an interactive proof assistant which implements Martin-Löf type theory[17], which could aids me to formalise a proof of the Halting problem. Because dependent types allows types to talk about values, the program written by those language could be encoded properties of values as types whose elements are proofs that the property is true, which means that a dependently typed programming language can be used as a logic, and is needed to be total, not crash or non-terminate. And mathematical proofs in **Agda** are written as structurally induction format, which are recursive functions that induce on some inductive type argument. Thus, by constructing some well-typed function that could finally terminate is equivalent to prove some mathematical proof. There for, **Agda** can be used as a framework to formalise formal logic systems and to prove the lemma which could be proved in mathematical.

1.3 Related Work

1.3.1 Proof By Contradiction

In 1936, Alan Turing has proved that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.[2]
The *halting problem* could be proved by contradictory. We could represent the decision problems as the set of objects that have the property in question.
Then the *halting problem* could be represent as halting set $H = \{(p, inp) |$ program p halts when run on input $inp\}$. Then we could prove that the following function h is not computable which means there is no total computable function which could

decide whether an arbitrary program p halts on arbitrary input inp[1]:
Suppose there exists a *Turing machine* which could decide the *halting problem*: $h(i,x) = \begin{cases} 1, & \text{if program } i \text{ halts on } x \\ 0, & \text{otherwise} \end{cases}$. Then we construct a universal *Turing machine* $u(x)$ which could take the binary code of another *Turing machine* as input such that inside u we run h on (x,x) and if the result of h is 1 then u will loop forever, otherwise u will halt. The last step is we run u on $\lfloor u \rfloor$ that is the binary code of the *Turing machine* u, which means inside u *Turing machine* h will run on $(\lfloor u \rfloor, \lfloor u \rfloor)$. Nevertheless, if the universal *Turing machine* u finally halt on its binary code, then the *Turing machine* h will return 1, nevertheless u will loop forever by the definition of *Turing machine u*. Thus, u can't halt on its binary code, however in which situation h will return 0 which means u will halt finally. By conclusion, there is no universal *Turing machine* that could decide the *halting problem* which proved by contradictory [2].

Chapter 2

Design and Implementation

2.1 WHILE Language Model in Agda

2.1.1 Tree Data Structure

The language **WHILE** computes with *trees* data structure built from a finite set. Thus we define a tree data structure $\mathbb{D}$ with several related functions in **Agda** at first. We must define the *atoms* for the *trees* at first. *Atoms* means they can't be divided further into subparts. However it is really complex to define a lot of *atoms* which may make our proof more complicated. In fact, we can define only one *atom* called *nil*, because any other values, or say other "atoms" we presumable to define could be constructed by combining different number of *nil* in different order. Thus, we define the data structure as:

```
data D : Set where
  dnil : D
  _•_  : D → D → D
```

And provide the approach to visit the first or the second element of an element in $\mathbb{D}$.

2.1.2 Syntax

The syntax of **WHILE** is defined following the syntax definition in the book[12] as:

2.1.2.1 Expression

An expression is construct in a binary tree format, which has the same format of the data structure we defined previously. Then an expression is either the value of some

variable, the *atom* value which is the *nil*, the first or the second value of another expression, the combination of two expression, or the equality of two expression. The definition of the syntax of *Expression* could be found in A.1.
Then we define the data type of $\mathbb{E}$ in **Agda** :

```
data E (n : ℕ) : Set where
  var  : Fin n → E n
  nil  : E n
  cons : E n → E n → E n
  hd   : E n → E n
  tl   : E n → E n
  _=?_ : E n → E n → E n
```

We use the member in a **finite set** to represent the variables instead of variable names. For example **Fin n** is a finite set that contain **n** elements from **zero** to $\underbrace{\textbf{suc (suc} \ldots \textbf{suc (zero))}}_{\textbf{n - 1}}$. Then we can directly use the element in the set to indicate variables name (**zero** is the first variable and so on).

2.1.2.2 Command

An *command* is either the assignment from some expression to some variable, or the sequence of two *commands*, or the *while* loop. The definition of the syntax of *Command* could be found in A.2.
Then we define the data type of $\mathbb{C}$ in **Agda** :

```
data C (n : ℕ) : Set where
  _:=_   : Fin n  → E n → C n
  _→→_ : C n → C n → C n
  while  : E n → C n → C n
```

2.1.2.3 Program

The program is consist of an *input* variable which is the variable to store the *input*, an *output* variable which is the variable to store the final result, and a *Command*:
Programs $\ni$ P ::= **read** X **;** C **; write** Y
And we can define the same data type of $\mathbb{P}$ in **Agda** :

```
data P (n : ℕ) : Set where
  prog : Fin n → C n → Fin n → P n
```

2.1.3 Semantics

To define the semantics of **WHILE** language, we must give a definition of *Partial Function* at first[12]:
Let A*,* B be sets, a partial function g is written as $g : A \to B_\perp$ and we said g is effectively computable if there is an effective procedure such that for any $x \in A$:

- The procedure eventually halts, yielding $g(x) \in B$, if $g(x)$ is defined;
- The procedure never halts, if $g(x)$ is undefined.

Then we could show that the program in **WHILE** can be used as a partial function from $\mathbb{D}$ to $\mathbb{D}_\perp$.

2.1.3.1 Environment

We should define the *environment* of the *command*, written as $[x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n]$ to indicate the finite mapping function such that $h(x_i) = v_i$, where $v_i \in \mathbb{D}$. Then we use the notion σ to indicate the *environment* in **WHILE** that has type *Store*, and for $p \in \mathbb{P}$, p = read X; C; write Y, the initial store σ_0^p is $[X \mapsto d, Y_1 \mapsto nil, \ldots, Y_n \mapsto nil]$, and $\forall$ variable X and Z such that X and Z are variables in program p and $X \neq Z$, then Z is in Y_i.
In **Agda** , we use the data type vector **Vec D n** to represent the store. Vector in **Agda** have the type:

```
data Vec {a} (A : Set a) : ℕ → Set
```

which bind a list of certain type element with certain number of length. Because the *store* and the program use the same **n** for both finite set of variable and its correspondence value, the program is impossible to meet the situation that one variable hasn't been defined.

2.1.3.2 Semantics of Expression

Then we can define the evaluation function ε with the type of $\mathbb{E} \longrightarrow (Store^p \to \mathbb{D})$, which means for $e \in \mathbb{E}$ and a given *store* of program $\mathbb{P}$: $\sigma \in Store^{\mathbb{P}}$, $\varepsilon[\![e]\!]\sigma = d \in \mathbb{D}$. The definition of the evaluation function could be find in A.3. The **Agda** implementation of the evaluation function is defined as following:

```
eval : {n : ℕ} → E n → Vec D n → D
eval (var x) v = dlookup x v
eval nil v = dnil
eval (cons e1 e2) v = eval e1 v • eval e2 v
eval (hd e) v = dhead (eval e v)
eval (tl e) v = dtail (eval e v)
eval (e1 =? e2) v with equalD? (eval e1 v) (eval e2 v)
eval (e1 =? e2) v | eq x = dnil • dnil
eval (e1 =? e2) v | neq x = dnil
```

2.1.3.3 Semantics of Command

The execution of a *Command* in the program $\mathbb{P}$ could be used as a function $f : \mathbb{C} \times Store^{\mathbb{P}} \times Store^{\mathbb{P}}$. However because we can't guarantee that the execution of a *Command* will eventually halt and yielding some output, the execution function should be a partial function $f : \mathbb{C} \times Store^{\mathbb{P}} \times Store^{\mathbb{P}}_{\perp}$. From this point of view, it is better to use a relation instead of a partial function to represent the execution of a *command* $c \in \mathbb{C}$ as $c \vdash \sigma \to \sigma' \subseteq \mathbb{C} \times Store^{\mathbb{P}} \times Store^{\mathbb{P}}$ where σ' is the new *environment* updated by the execution of *command* c. The definition of the execution *command* relationship could be find in A.4. The **Agda** implementation of the execution relationship is defined as following:

```
data _⊢_⇒_ {n : ℕ} : C n → Vec D n  → Vec D n → Set where
  whilef : {e : E n}{c : C n}{env : Vec D n}
         → isNil (eval e env)
         → (while e c) ⊢ env ⇒ env
  whilet : {e : E n}{c : C n}{env1 env2 env3 : Vec D n}
         → isTree (eval e env1)
         → c ⊢ env1 ⇒ env2
         → (while e c) ⊢ env2 ⇒ env3
         → (while e c) ⊢ env1 ⇒ env3
  assign : {v : Fin n}{e : E n}{env : Vec D n}
         → (v := e) ⊢ env ⇒ (updateE v (eval e env) env)
  seq    : {c1 c2 : C n}{env1 env2 env3 : Vec D n}
         → c1 ⊢ env1 ⇒ env2
         → c2 ⊢ env2 ⇒ env3
         → (c1 →→ c2) ⊢ env1 ⇒ env3
```

2.1.3.4 Semantics of Program

Similar to the definition of execution of $\mathbb{C}$ in **WHILE** , the execution of program $\mathbb{P}$ should also be defined as a relationship. Following the definition given by *Jones, Neil D* in his paper[12], we could know that the semantics of **WHILE** is:

$[\![\bullet]\!]^{\text{WHILE}} : \mathbb{P} \to (\mathbb{D} \to \mathbb{D}_\bot)$ defined for p = read X; C; write Y by:
$[\![p]\!]^{\text{WHILE}} = e$ if $C \vdash \sigma_0^p(d) \to \sigma$ and $\sigma(Y) = e$
If there is no e such that $[\![p]\!] = e$, then p **loops** on d; otherwise p terminates on d.
Following the definition, we define the partial relationship in **Agda** as follow:

```
data ExecP {n : ℕ} : P n → D → D → Set where
  terminate : (x y : Fin n){c : C n}{env : Vec D n}{d : D}
            → c ⊢ (updateE x d initialVec) ⇒ env
            → ExecP (prog x c y) d (dlookup y env)
```

The example of **WHILE** program and the execution of **WHILE** program could be find at A.5.

2.1.4 Run WHILE in K Steps

Even the execution of the command $\mathbb{C}$ and the program $\mathbb{P}$ are both a partial relationship, which means we can't guarantee the command or the program will eventually halt on some input(that is what we are proving), we can still define the partial function that try to execute the command $\mathbb{C}$ and program $\mathbb{P}$ in k time.
Firstly we should define some data type to recored the execution step number of a given command.

```
record ResultCT {n : ℕ}(c : C n)(inp : Vec D n) : Set where
  field
    out  : Vec D n
    exe  : c ⊢ inp ⇒ out
    time : ℕ
```

Then we can construct the function to prove that one command may be executed in k time:

```
kStepC : {n : ℕ} → (k : ℕ) → (c : C n) → (inp : Vec D n)
       → (Maybe (ResultCT c inp))
```

The idea of that function is to do induction on the k at first. None command could be run in *zero* step. Then the function will do induction on the *command.* The assignment step will only cost *1* step. The steps costed on the sequence of two *command* c_1 and c_2 will be the sum of steps that costed on c_1 and the steps that costed on c_2. Similarly in the *while* loop, the *command* will cost *zero* step if the

expression to the *while* loop could be evaluated to *false*. It will cost steps that costed on c and the continuous steps that costed on the following *while* loop as the total steps to the *command*.
Finally, if a command c could be executed in k steps, then a program $p =$ (read X; c; write Y) could also be executed in k steps:

```
kStepP : {n : ℕ} → (time : ℕ) → (p : P n) → (inp : D)
        → (Maybe (? D (ExecP p inp)))
```

2.2 Universal WHILE model

2.2.1 Interpret WHILE program

In order to construct the universal **WHILE** model in **Agda** later, we must define the method to code a program into $\mathbb{D}$ in order to feed the program as the input to the universal **WHILE** program later. It is important to define the operator $\bullet$ of our data structure $\mathbb{D}$ with no association, which would avoid the ambiguous.
Initially we should define some constants to indicate some distinct elements of $\mathbb{D}$. Those constants would represent the program in format of $\mathbb{D}$. The definition could be found in B.1.
Then we defined the function that map the program to $\mathbb{D}$: $\lfloor \bullet \rfloor \in \mathbb{P} \times \mathbb{D}$. The *code* function is consisted of three parts.

2.2.1.1 Code the Expression

Initially we should define the function that code the *expression* into $\mathbb{D}$. The definition of the mapping function of *expression* could be find in B.2. Then we could define the function in **Agda** following the same definition:

```
codeE : {n : ℕ} → E n → D
codeE (var x) = dvar • dftod x
codeE nil = dquote • dnil
codeE (cons e₁ e₂) = dcons • (codeE e₁ • codeE e₂)
codeE (hd e) = dhd • codeE e
codeE (tl e) = dtl • codeE e
codeE (e₁ =? e₂) = d=? • (codeE e₁ • codeE e₂)
```

2.2.1.2 Code the Command

Then we should define the function that code the *command* into 𝔻. The definition of the mapping function of *command* could be find in B.2. Then we could define the function in **Agda** following the same definition:

```
codeC : {n : ℕ} → C n → D
codeC (x := e) = d:= • ((dvar • dftod x) • codeE e)
codeC (c₁ →→ c₂) = d→→ • (codeC c₁ • codeC c₂)
codeC (while e c) = dwhile • (codeE e • codeC c)
```

2.2.1.3 Code the Program

Finally we should define the function that code the *program* into 𝔻. In addition to follow the mapping function, I also add the number of variable of the program into the result of coding. The definition of the mapping function of *program* could be find in B.2. Then we could define the function in **Agda** following the same definition:

```
codeP : {n : ℕ} → P n → D
codeP {n} (prog x c y) = const n • ((dvar • dftod x) •
                                     (codeC c •
                                     (dvar • dftod y)))
```

2.2.1.4 Decode

Beyond the coding method that map the *program* to 𝔻, I also define the function that decode the 𝔻 and map it to *program*. However because the function *decode* is a partial function, sometimes it may cause decoding fail because the input 𝔻 doesn't following the format of program.

```
decodeE : {n : ℕ} → D → Maybe (E n)
decodeC : {n : ℕ} → D → Maybe (C n)
decodeP : D → Maybe (? ℕ P)
```

2.2.2 Universal WHILE model

Initially we should define the variable in the **WHILE** language, the definition could be found in B.3.
Then we should define some syntax sugar such as *if* and *if-else*:

```
if8 : E 8 → C 8 → C 8
if8 e c = (Z := e) →→ while (var Z) ((Z := nil) →→ c)

if-else8 : E 8 → C 8 → C 8 → C 8
if-else8 e c₁ c₂ = (Z := e) →→
                   (W := cons nil nil) →→
                   ((while (var Z)
                           ((Z := nil) →→
                           ((W := nil) →→
                           c₁))) →→
                   (while (var W)
                          ((W := nil) →→
                         c₂)))
```

Finally we should define the universal **WHILE** program. Here we firstly define the universal **WHILE** program that could simulate other **WHILE** program which has only one variable. The definition could be found in B.4.
The the program is defined in **Agda** as:

```
universalI : P 8
universalI = prog PD ((Pp := hd (var PD))
                      →→
                      (Cc := hd (tl (var Pp)))
                      →→
                      (Cd := cons (var Cc) nil)
                      →→
                      (St := nil)
                      →→
                      (V1 := tl (var PD))
                      →→
                      (while (var Cd) STEP-I))
                   V1
```

2.2.2.1 Interpret by Agda

We can imitate the simulation step following the definition using **Agda** at first. The definition of the *STEP* Macro could be find in B.5.
Initially we could define the data relationship $(Cd, St, V1) \Rightarrow (Cd', St', V1') \in (\mathbb{D}, \mathbb{D}, \mathbb{D}) \times (\mathbb{D}, \mathbb{D}, \mathbb{D})$ as a one step relationship.

```
data _⇒_ : D × D × D → D × D × D → Set where
  equote  : (d Cr St V1 : D)
            → < (dquote • d) • Cr , St , V1 >
            ⇒ < Cr , d • St , V1 >
  evar1   : (Cr St V1 : D)
            → < (dvar • dftod {1} zero) • Cr , St , V1 >
            ⇒ < Cr , V1 • St , V1 >
  ehd     : (E Cr St V1 : D)
            → < (dhd • E) • Cr , St , V1 >
            ⇒ < E • (dohd • Cr) , St , V1 >
  edohd   : (T Cr St V1 : D)
            → < dohd • Cr , T • St , V1 >
            ⇒ < Cr , (dfst T) • St , V1 >
  etl     : (E Cr St V1 : D)
            → < (dtl • E) • Cr , St , V1 >
            ⇒ < E • (dotl • Cr) , St , V1 >
  edotl   : (T Cr St V1 : D)
            → < dotl • Cr , T • St , V1 >
            ⇒ < Cr , (dsnd T) • St , V1 >
  econs   : (E₁ E₂ Cr St V1 : D)
            → < (dcons • (E₁ • E₂)) • Cr , St , V1 >
            ⇒ < E₁ • (E₂ • (docons • Cr)) , St , V1 >
  edocons : (U T Cr St V1 : D)
            → < docons • Cr , U • (T • St) , V1 >
            ⇒ < Cr , (T • U) • St , V1 >
  e=?     : (E₁ E₂ Cr St V1 : D)
            → < (d=? • (E₁ • E₂)) • Cr , St , V1 >
            ⇒ < E₁ • (E₂ • (do=? • Cr)) , St , V1 >
  edo=?   : (U T Cr St V1 : D)
            → < do=? • Cr , U • (T • St) , V1 >
            ⇒ < Cr , (dequal T U) • St , V1 >
  e→→       : (C₁ C₂ Cr St V1 : D)
            → < (d→→ • (C₁ • C₂)) • Cr , St , V1 >
            ⇒ < C₁ • (C₂ • Cr) , St , V1 >
  e:=     : (E Cr St V1 : D)
            → < (d:= • ((dvar • dftod {1} zero) • E)) • Cr , St , V1 >
            ⇒ < E • (doasgn • Cr) , St , V1 >
  edoasgn : (W Cr St V1 : D)
            → < doasgn • Cr , W • St , V1 >
            ⇒ < Cr , St , W >
  ewhile  : (E C Cr St V1 : D)
            → < (dwhile • (E • C)) • Cr , St , V1 >
            ⇒ < E • (dowh • ((dwhile • (E • C)) • Cr)) , St , V1 >
  edowhf  : (E C Cr St V1 : D)
            → < dowh • ((dwhile • (E • C)) • Cr) , dnil • St , V1 >
```

```
          ⇒ < Cr , St , V1 >
  edowht  : (E C X Y Cr St V1 : D)
          → < dowh • ((dwhile • (E • C)) • Cr) , (X • Y) • St , V1 >
          ⇒ <  C • ((dwhile • (E • C)) • Cr)  , St , V1 >
  enil    : (St V1 : D) → < dnil , St , V1 > ⇒ < dnil , St , V1 >
```

Then we should define the several steps relationship $(Cd, St, V1) \Rightarrow^* (Cd', St', V1')$ $\in (\mathbb{D}, \mathbb{D}, \mathbb{D}) \times (\mathbb{D}, \mathbb{D}, \mathbb{D})$.

```
data _⇒*_ : D × D × D → D × D × D → Set where
  id   : (Cr St V1 : D) → < Cr , St , V1 > ⇒* < Cr , St , V1 >
  seq  : (Cr₁ Cr₂ Cr₃ St₁ St₂ St₃ V1₁ V1₂ V1₃ : D)
         → < Cr₁ , St₁ , V1₁ > ⇒  < Cr₂ , St₂ , V1₂ >
         → < Cr₂ , St₂ , V1₂ > ⇒* < Cr₃ , St₃ , V1₃ >
         → < Cr₁ , St₁ , V1₁ > ⇒* < Cr₃ , St₃ , V1₃ >
```

We should proof the associative of relation $\Rightarrow^*$.

```
  ⇒*-m : (Cr₁ Cr₂ Cr₃ St₁ St₂ St₃ V1₁ V1₂ V1₃ : D)
              → < Cr₁ , St₁ , V1₁ > ⇒* < Cr₂ , St₂ , V1₂ >
              → < Cr₂ , St₂ , V1₂ > ⇒* < Cr₃ , St₃ , V1₃ >
              → < Cr₁ , St₁ , V1₁ > ⇒* < Cr₃ , St₃ , V1₃ >
  ⇒*-b : (Cr₁ Cr₂ Cr₃ St₁ St₂ St₃ V1₁ V1₂ V1₃ : D)
              → < Cr₁ , St₁ , V1₁ > ⇒* < Cr₂ , St₂ , V1₂ >
              → < Cr₂ , St₂ , V1₂ > ⇒  < Cr₃ , St₃ , V1₃ >
              → < Cr₁ , St₁ , V1₁ > ⇒* < Cr₃ , St₃ , V1₃ >
```

Then we can prove that if for $E \in \mathbb{E}$, $\varepsilon[\![E]\!][V1 \mapsto d] = d_e$, then $((codeEE \cdot Cr), St, d) \Rightarrow^* (Cr, (d_e \cdot St), d)$.

```
⇒*e : (e : E 1) → (d₁ d₂ Cr St : D)
       → eval e (updateE zero d₁ initialVec) ≡ d₂
       → < codeE e • Cr , St , d₁ > ⇒* < Cr , d₂ • St , d₁ >
```

After that, we can prove that if for $C \in \mathbb{C}$, $C \vdash [V1 \mapsto d_1] \Rightarrow [V1 \mapsto d_2]$, then $((codeCC \cdot Cr), St, d_1) \Rightarrow^* (Cr, St, d_2)$.

```
⇒*ok : (c : C 1) → (d₁ d₂ Cr St : D) → (out : Vec D 1)
       → c ⊢ updateE zero d₁ initialVec ⇒ out
       → dlookup zero out ≡ d₂
       → < codeC c • Cr , St , d₁ > ⇒* < Cr , St , d₂ >
```

This proof means the execution of *command* has relationship with the relation $\Rightarrow$, which means one step of execution of *command* is corresponding to the several step $\Rightarrow^*$, which is the simulation of **WHILE** program in **Agda**.

2.2.2.2 Interpret by WHILE program

Then we should define real universal **WHILE** program. The most important part is the *STEP* Macro. Before we defining the Macro, we should define some constants in $\mathbb{E}$ which could be found in B.6.
Then we can define *STEP* as a *command* in **Agda** :

```
STEP-I : C 8
STEP-I =  the interpretation command
          following the definition of STEP Macro
          and the syntax of WHILE program
```

Then we can prove that the simulation hold by **Agda** has one step correspondence with the Macro *STEP*. That is, if $(Cd, St, V1) \Rightarrow (Cd', St', V1')$, then *STEP* $\vdash [\ldots, Cd, St, V1, \ldots] \Rightarrow [\ldots, Cd', St', V1', \ldots]$.

```
  c-h : {Pd P C : D}(d₁ d₂ Cr₁ Cr₂ St₁ St₂ : D)
        → < Cr₁ , St₁ , d₁ > ⇒ < Cr₂ , St₂ , d₂ >
        → STEP-I ⊢ (Pd :: P :: C :: Cr₁ :: St₁ :: d₁ :: dnil :: dnil :: [])
                 ⇒ (Pd :: P :: C :: Cr₂ :: St₂ :: d₂ :: dnil :: dnil :: [])
```

Because both the relation $\Rightarrow$ and the execution of *while* loop does the induction on one step, we could prove the several steps correspondence between **Agda** simulation and the Macro *STEP*.

```
step-I-ok : (c : C 1) → (d₁ d₂ : D)
          → < codeC c • dnil , dnil , d₁ > ⇒* < dnil , dnil , d₂ >
          → while (var Cd) STEP-I ⊢ (codeP (prog zero c zero)) • d₁ ::
                                    (codeP (prog zero c zero)) ::
                                    codeC {1} c :: codeC {1} c • dnil  ::
                                    dnil :: d₁ :: dnil :: dnil :: [])
                                  ⇒ (codeP (prog zero c zero)) • d₁ ::
                                    (codeP (prog zero c zero)) ::
                                    codeC {1} c :: dnil :: dnil ::
                                    d₂ :: dnil :: dnil :: [])
```

From this proof we can know that if **Agda** could simulate some **WHILE** program, then the universal **WHILE** program can simulate the same **WHILE** program.

2.2.3 Correctness of Universal WHILE model

Finally by using the proof in the two previous parts, we can prove the correctness of the universal **WHILE** program.

```
execP-uni :  (p : P 1) → (d₁ d₂ : D)
             → ExecP P d₁ d₂
             → ExecP universalI ((codeP p)● d₁) d₂
```

As a result, we can conclude that for $p \in \mathbb{P}$ and inp, $out \in \mathbb{D}$, if $p(inp) \equiv output$, then the universal **WHILE** program u, $u(\lfloor p \rfloor \bullet inp) \equiv output$

2.3 Proof to Halting Problem

2.3.1 Construct WHILE program U

To prove the undecidability of *halting problem* by contradiction, we should construct a program at first. Following the definition on *wiki*[2] and on paper[7], we could construct a program **U**. The strategy to construct **U** is decribed in 1.3.1. However by considering the *syntax* and *semantic* of **WHILE** program and universal **WHILE** program, we know that we must feed the code of another program into our universal program as part of the argument. That is, when we are constructing the program **U**, and assume there is a program h that could decide the halting problem, then the argument to the program **U** should be $(\lfloor h \rfloor \bullet input)$. And to unify the argument to the program, the program h inside **U** should run on $(input \bullet (\lfloor h \rfloor \bullet input))$. Thus, the definition of program **U** in **Agda** is:

```
U : P 8
U = prog PD ((Pp := hd (var PD))
            →→
            (Cc := hd (tl (var Pp)))
            →→
            (Cd := cons (var Cc) nil)
            →→
            (St := nil)
            →→
            (V1 := cons (tl (var PD)) (cons  (var Pp) (tl (var PD))))
            →→
            (while (var Cd) STEP-I)
            →→
```

```
      if-else8 (var V1) (while (cons nil nil) (V1 := var V1))
                        (V1 := var V1)
      )
    V1
```

Because we don't have empty *command*, we use the command x :=var x, assign the same value to its original variable, to indicate the empty *command.*

2.3.1.1 Property 1 of U

From out definition of the program **U**, we can prove that if the execution result of $h(input \bullet (\lfloor h \rfloor \bullet input))$ is *true*, then if we feed h to program **U** and execute program **U** on $(\lfloor h \rfloor \bullet input)$, program **U** will never terminate.
Initially we can prove that the infinite loop can't terminate, and if there is *command* in format of *while true command*, then this *command* is an infinite loop.

```
wt : {n : ℕ} → C n → C n
wt c = while (cons nil nil) c


wt-loop : {t : D}{n : ℕ}{c : C n}{env₁ env₂ : Vec D n}
        → (p : wt c ⊢ env₁ ⇒ env₂) → loop-ct p ≡ t → ⊥
wt-loop (whilef ()) x
wt-loop {dnil} (whilet x p p₁) ()
wt-loop {.(loop-ct p) • .(loop-ct p₁)} (whilet x p p₁) refl
       = wt-loop {loop-ct p₁} p₁ refl
```

Here we use *nil* to indicate *false* in **WHILE** and *others* to indicate *true* in **WHILE** program. The proof function does induction on the *call tree*, which means the *assignemnt* is the leaf of the tree, *sequence* and *while* loop both has two branches.
Then we can prove that for any $h \in \mathbb{P}$, if $h(input \bullet (\lfloor h \rfloor \bullet input))$ yielding *true*, then the execution of **U** on $(\lfloor h \rfloor \bullet input)$ will never terminate.

```
execP-U-loop :  {h : P 1} → (d₁ d₂ : D)
              → ExecP h (d₁ • ((codeP h) • d₁)) d₂
              → (d₂ ≡ dnil → ⊥)
              → (∀ {d₃ : D} → ExecP U ((codeP h) • d₁) d₃ → ⊥)
```

2.3.1.2 Property 2 of U

From out definition of the program **U**, we can prove that if the execution result of $h(\mathit{input} \bullet (\lfloor h \rfloor \bullet \mathit{input}))$ is *false*, then if we feed h to program **U** and execute program **U** on $(\lfloor h \rfloor \bullet \mathit{input})$, program **U** will terminate immediately.

```
execP-U-halt :  {h : P 1} → (d₁ d₂ : D)
               → ExecP h (d₁ • ((codeP h) • d₁)) d₂
               → d₂ ≡ dnil
               → ExecP U ((codeP h) • d₁) d₂
```

2.3.2 Proof the Undecidability of Halting Problem

Finally we assume that there exists some program h that will decide the *halting problem* following the definition in 1.2.3.

2.3.2.1 Property 1 of Machine H

The program h is a program of **WHILE** that has the property of, for all $p \in \mathbb{P}$ and $\mathit{input} \in \mathbb{D}$, if p ***halt*** on *inp*, then $h\ (\lfloor p \rfloor \bullet \mathit{input}))$ yielding *true*.

```
prop₁ : ∀ {n : ℕ} → ∀ {p : P n} → ∀ {inp : D}
        → (? D (ExecP p inp)
        → ExecP h ((codeP p) • inp) dtrue)
```

2.3.2.2 Property 2 of Machine H

The program h also has the property of, for all $p \in \mathbb{P}$ and $\mathit{input} \in \mathbb{D}$, if p doesn't ***halt*** on *inp*, then $h\ (\lfloor p \rfloor \bullet \mathit{input}))$ yielding *false*.

```
prop₂ : ∀ {n : ℕ} → ∀ {p : P n} → ∀ {inp : D}
        → (∀ {out : D} → ExecP p inp out → ⊥)
        → ExecP h ((codeP p) • inp) dfalse
```

2.3.2.3 Propositional Proof

Then we can abstract the proof from the two properties of program **U** and the program h. We can name the property "**U** *halt* on $(\lfloor h \rfloor \bullet \mathit{input})$" as X, "$h\ (\lfloor p \rfloor \bullet \mathit{input}))$ yielding *true*" as Y and "$h\ (\lfloor p \rfloor \bullet \mathit{input}))$ yielding *false*" as Z. Then we can rename the two properties of **U** as xy and nxz, and the two properties of h as ynx and zx. Note that $\neg\exists x, Px \equiv \forall x, \neg Px$. Then we can get contradiction from those four propositions.

```
postulate
  X Y Z : Set
  xy   : X → Y
  nxz  : (X → ⊥) → Z
  ynx  : Y → X → ⊥
  zx   : Z → X

a⊥ : X → ⊥
a⊥ a = ynx (xy a) a

bot : ⊥
bot = a⊥ (zx (nxz a⊥))
```

2.3.2.4 Final Proof

Finally we can prove the *undecidability* of *halting problem* by contraction, which means we assume there exists a program *h* which could *decide* the *halting problem* and prove the contradiction.

```
halt-contradiction : {h : P 1}
                   → (∀ {n : ℕ} → ∀ {p : P n} → ∀ {inp : D}
                      → (? D (ExecP p inp)
                          → ExecP h ((codeP p) • inp) dtrue)
                      ×' ((∀ {out : D} → ExecP p inp out → ⊥)
                          → ExecP h ((codeP p) • inp) dfalse))
                   → ⊥
halt-contradiction {h} p = exec-U-⊥ (dnil ,
                                      (execP-U-halt {h} ((codeP U)) dnil
                                        (u-loop
                                          (lambda {out} q
                                              → exec-U-⊥ (out , q)))
                                        refl))
  where
    prop = p {8}{U}{((codeP h) • (codeP U))}

    u-halt : ? D (ExecP U ( (codeP h) • (codeP U)))
        → ExecP h ( (codeP U) • ((codeP h) • (codeP U))) dtrue
    u-halt = proj₁ prop

    u-loop : (∀ {out : D} → ExecP U ((codeP h) •  (codeP U)) out → ⊥)
        → ExecP h ( (codeP U) • ( (codeP h) •  (codeP U))) dfalse
    u-loop = proj₂ prop

    exec-U-⊥ : ? D (ExecP U ((codeP h) • (codeP U))) → ⊥
    exec-U-⊥ (d , p) = execP-U-loop ((codeP U)) dtrue
                        (u-halt (d , p)) (lambda { () }) p
```

Chapter 3

Evaluation, Summary and Futher Work

3.1 Evaluation to the Project

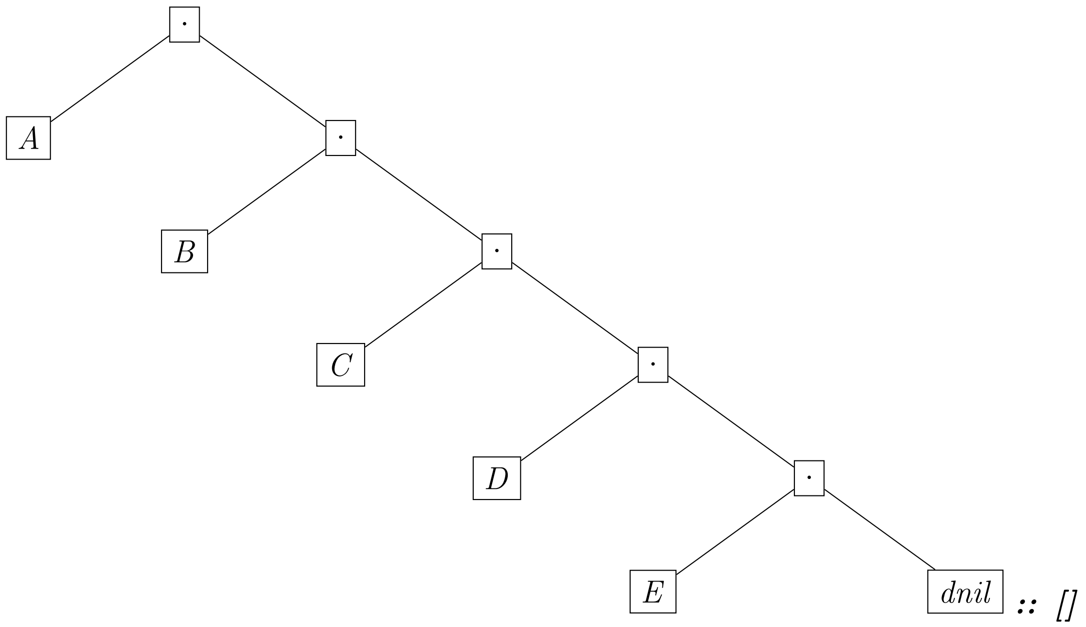
3.2 Summary and Conclusion

3.3 Future Work

3.3.1 WHILE Program Variables Transformation

The universal **WHILE** program defined in could only simulate the **WHILE** program that has only one variable. Thus the final proof could only assume a machine h which has only one variable that could *decide* the *halting problem.* However the **WHILE** program that has only one variable (we name it as **WHILE-I** program) has the same computation ability compare to the **WHILE** program that has many variables. That means, the number of variable to the **WHILE** program doesn't matter, and doesn't violate the property of *Turing-completeness* to the **WHILE** program. However, because **Agda** has strict type, the program must know its variable number before it has been defined. Thus, we can construct some rules to transform the **WHILE** program to **WHILE-I** program and prove that they have the same effect, which means for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, $p(input)$ yielding *output* implies that $\exists\ p - I \in \mathbb{P}\ 1$, $p - I(input)$ yielding *output*.
Initially we should transform the *environment* to the program from **Vec D n** to **Vec D 1**, which means accumulate all the variables in the first *environment* to the first element in the second *environment* by the operator $\cdot$. For example, initially we have

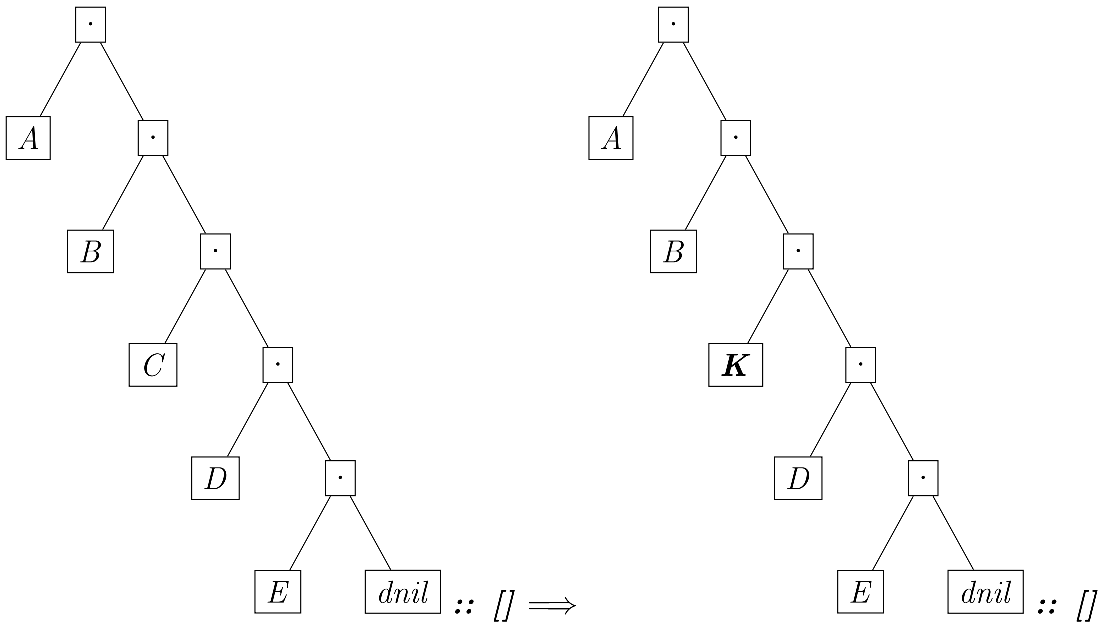
environment of $(A :: B :: C :: D :: E ::[])$, then the transformation will construct a tree structure data for the first element of the new *environment*:



Then we can transform the *expression*. For the *expression* of *head*, *tail*, *cons*, *nil* and *equality*, we can easily recursively transform the target *expression* from the argument. For the *expression* that use the value of variable that in the *environment*, for example *var* C in our previous example, we can use the *expression head (tail (tail (var zero)))* to get the same value from the transformed *1* variable environment.
Then similarly we can transform the *command*. For the *command* of *sequence* and *while* loop, we can easily recursively transform the target *command* from the argument. For the *expression* of *assignment*, for example we assign variable C to $K \in \mathbb{E}$, the *environment* with five variables updated as:

$$(A :: B :: C :: D :: E ::[]) \Longrightarrow (A :: B :: \boldsymbol{K} :: D :: E ::[])$$

Then we can write the new *assignment command* as
$zero := (hd\ (var\ zero)) \cdot ((hd\ (tl\ (var\ zero))) \cdot ((K) \cdot (tl\ (tl\ (tl\ (var\ zero))))))$ and



Finally we can transform the *program*. We should transform the initial *environment* at first. Then we can transform the *command*. Finally we should get the result from the transformed *environment*.
The proof of the correctness of the transformation will be done in the future.

3.3.2 Interpret WHILE Program with Arbitrary Variables

If we can prove that the **WHILE** program has the same computation ability with **WHILE-I** program which has only one variable, we could conclude that our universal **WHILE** program defined in 3.3.1 could simulate the **WHILE** program has arbitrary variables. For example, for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, if we want to use our universal **WHILE** program to simulate $p(input)$, then we could transform p to $p - I$ which has only one variable by our predefined transformation function. And we know that $p(input)$ yielding *output* implies that $p - I(input)$ yielding *output* by the proof of correctness of the transformation function. And by the proof of the correctness of the universal **WHILE** program we know that $p - I(input)$ yielding *output* implies that $u(\lfloor p - I \rfloor \cdot input)$ yielding *output*. Thus we can conclude that for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, $p(input)$ yielding *output* implies $u(\lfloor p - I \rfloor \cdot input)$ yielding *output* which means universal **WHILE** program defined in 3.3.1 could simulate the **WHILE** program has arbitrary variables.
Thus for the proof of *halting problem*, we could say that for $p \in \mathbb{P}\ n$, h could decide the *halting problem* implies *false*.

Appendix A

Definition of WHILE Language

The definition of **WHILE** language used in the project are following the definition in the paper *Computability and complexity: from a programming perspective* by *Jones, Neil D* in 1997[12].

A.1 Syntax of Expression

$$
\begin{array}{llll}
\text{Expressions} \ni & \text{E, F} ::= & \text{X} & (\text{for X} \in \text{Vars}) \\
& & \mid d & (\text{for atom } d\text{, one atom } nil \text{ defined in } \textbf{Agda}\ ) \\
& & \mid \textbf{cons}\ \text{E F} & \\
& & \mid \textbf{hd}\ \text{E} & \\
& & \mid \textbf{tl}\ \text{E} & \\
& & \mid \textbf{=?}\ \text{E F} &
\end{array}
$$

A.2 Syntax of Command

$$
\begin{array}{lll}
\text{Commands} \ni & \text{C, D} ::= & \text{X := E} \\
& & \mid \text{C}\ \textbf{;}\ \text{D} \\
& & \mid \textbf{while}\ \text{E}\ \textbf{do}\ \text{C}
\end{array}
$$

A.3 Semantics of Expression

The definition of evaluation function ε is: for $e \in \mathbb{E}$ and a given *store* of program $\mathbb{P}$, $\sigma \in Store^{\mathbb{P}}$, $\varepsilon[\![e]\!]\sigma = d \in \mathbb{D}$.

$$\begin{array}{lcl}
\varepsilon]\!]\mathrm{X}]\!]\sigma & = & \sigma(\mathrm{X}) \\
\varepsilon[\![\mathrm{d}]\!]\sigma & = & \mathrm{d} \\
\varepsilon[\![\mathrm{cons\ E\ F}]\!]\sigma & = & \varepsilon[\![\mathrm{E}]\!]\sigma \,.\, \varepsilon[\![\mathrm{F}]\!]\sigma \\
\varepsilon[\![\mathrm{hd\ E}]\!]\sigma & = & \begin{cases} e & \text{if } \varepsilon[\![E]\!]\sigma = (e, f) \\ \mathrm{nil} & \text{otherwise} \end{cases} \\
\varepsilon[\![\mathrm{tl\ E}]\!]\sigma & = & \begin{cases} f & \text{if } \varepsilon[\![E]\!]\sigma = (e, f) \\ \mathrm{nil} & \text{otherwise} \end{cases} \\
\varepsilon[\![=?\ \mathrm{E\ F}]\!]\sigma & = & \begin{cases} \text{true} & \text{if } \varepsilon[\![E]\!]\sigma = \varepsilon[\![\mathrm{F}]\!]\sigma \\ \text{false} & \text{otherwise} \end{cases}
\end{array}$$

A.4 Semantics of Command

The definition of the execution relationship is: for $c \in \mathbb{C}$, $c \vdash \sigma \to \sigma' \subseteq \mathbb{C} \times \mathit{Store}^{\mathbb{P}} \times \mathit{Store}^{\mathbb{P}}$ where σ' is the new *environment* updated by the execution of command c.

$$\begin{array}{lll}
\mathrm{X{:=}E} \vdash \sigma \to \sigma[\mathrm{X} \mapsto \mathrm{d}] & \text{if} & \varepsilon[\![E]\!]\sigma = \mathrm{d} \\
\mathrm{C;D} \vdash \sigma \to \sigma'' & \text{if} & \mathrm{C} \vdash \sigma \to \sigma' \text{ and } \mathrm{D} \vdash \sigma' \to \sigma'' \\
\textbf{while } \mathrm{E} \textbf{ do } \mathrm{C} \vdash \sigma \to \sigma'' & \text{if} & \mathrm{C}\ \varepsilon[\![E]\!]\sigma \neq \mathrm{nil},\ \mathrm{C} \vdash \sigma \to \sigma', \\
 & & \qquad \textbf{while } \mathrm{E} \textbf{ do } \mathrm{C} \vdash \sigma' \to \sigma'' \\
\textbf{while } \mathrm{E} \textbf{ do } \mathrm{C} \vdash \sigma \to \sigma & \text{if} & \mathrm{C}\ \varepsilon[\![E]\!]\sigma = \mathrm{nil}
\end{array}$$

A.5 Example WHILE program

Here we give an example code on **WHILE** language and use the defined part to simulate the program.

A.5.1 Example WHILE program in WHILE

The **WHILE** program **concat** which could concatenate two list into one define as below:

```
read X; (* X is (d.e) *)
  A := hd X; (* A is d *)
  Y := tl X; (* Y is e *)
  B := nil; (* B becomes d reversed *)
  while A do
```

```
    B := cons (hd A) B;
    A := tl A;
  while B do
    Y := cons (hd B) Y;
    B := tl B;
write Y
```

A.5.2 Example WHILE program in Agda

Here we construct the same program using the definition we defined in **Agda**, it should be the following format:

```
append : P 4
append = prog zero
             ((suc (suc zero) := hd (var zero))
             →→
             (suc zero := tl (var zero))
             →→
             (suc (suc (suc zero)) := nil)
             →→
             (while
               (var (suc (suc zero)))
               ((suc (suc (suc zero)) :=
                        cons (hd (var (suc (suc zero))))
                             (var (suc (suc (suc zero)))))
               →→
               ((suc (suc zero)) := tl (var (suc (suc zero))))))
               →→
             (while
               (var (suc (suc (suc zero))))
               ((suc zero := cons (hd (var (suc (suc (suc zero)))))
                                  (var (suc zero)))
             →→
               (suc (suc (suc zero)) := tl (var (suc (suc (suc zero)))))))
             (suc zero)
```

A.5.3 Execution of the example WHILE program in Agda

To run the program, we define three list (in format of $\mathbb{D}$) in which **list1** and **list2** are the two input lists and **list3** is the result:

```
list1 : D
list1 = ltod (1 :: 2 :: 3 :: [])
```

```
list2 : D
list2 = ltod (4 :: 5 :: 6 :: [])

list3 : D
list3 = ltod (1 :: 2 :: 3 :: 4 :: 5 :: 6 :: [])
```

Now we could execute the **WHILE** program using our definitions of syntax and semantics:

```
runAppend : ExecP append (list1 • list2) list3
runAppend = terminate zero (suc zero)
                {env = list1 • list2 :: list3 :: dnil :: dnil :: []}
            (seq {env1 = list1 • list2 :: dnil :: dnil :: dnil :: []}
             assign
            (seq {env1 = list1 • list2 :: dnil :: list1 :: dnil :: []}
             assign
            (seq {env1 = list1 • list2 :: list2 :: list1 :: dnil :: []}
             assign
            (seq {env1 = list1 • list2 :: list2 :: list1 :: dnil :: []}
                 {env2 = result}
                 {env3 =  list1 • list2 :: list3 :: dnil :: dnil :: []}
            (whilet {env1 = list1 • list2 :: list2 :: list1 :: dnil :: []}
                    {env2 = list1 • list2 :: list2 ::
                            dsnd list1 :: dfst list1 • dnil :: []}
                    {env3 = result}
             tt
            (seq assign assign)
            (whilet {env1 = list1 • list2 :: list2 ::
                            dsnd list1 :: dfst list1 • dnil :: []}
                    {env2 = list1 • list2 :: list2 :: dsnd (dsnd list1) ::
                            dfst (dsnd list1) • (dfst list1 • dnil) :: []}
                    {env3 = result}
             tt
            (seq assign assign)
            (whilet {env1 = list1 • list2 :: list2 :: dsnd (dsnd list1) ::
                            dfst (dsnd list1) • (dfst list1 • dnil) :: []}
                    {env2 = result}
                    {env3 = result}
             tt
            (seq assign assign)
            (whilef tt))))
            (whilet {env1 = result}
                    {env2 = list1 • list2 :: const 3 • list2 ::
```

```
                dnil :: ltod (2 :: 1 :: []) :: []}
        {env_3 = list1 • list2 :: list3 :: dnil :: dnil :: []}
 tt
(seq assign assign)
(whilet {env_1 = list1 • list2 :: const 3 • list2 ::
                dnil :: ltod (2 :: 1 :: []) :: []}
        {env_2 = list1 • list2 :: const 2 • (const 3 • list2) ::
                dnil :: ltod (1 :: []) :: []}
        {env_3 = list1 • list2 :: list3 :: dnil :: dnil :: []}
 tt
(seq assign assign)
(whilet {env_1 = list1 • list2 :: const 2 • (const 3 • list2) ::
                dnil :: ltod (1 :: []) :: []}
        {env_2 = list1 • list2 :: list3 :: dnil :: dnil :: []}
        {env_3 = list1 • list2 :: list3 :: dnil :: dnil :: []}
 tt
(seq assign assign)
(whilef tt))))))))
where
   result : Vec D 4
   result = list1 • list2 :: list2 ::
            dnil :: ltod (3 :: 2 :: 1 :: []) :: []
```

Appendix B

Definition in the Universal WHILE model

The definition of **WHILE** language used in the project are following the definition in the paper *Computability and complexity: from a programming perspective* by *Jones, Neil D* in 1997[12].

B.1 Constant in WHILE

First of all, we define a function that would construct some value in data of $\mathbb{D}$ based on the natural number $\mathbb{N}$:

```
const : (n : ℕ) → D
const zero = dnil
const (suc n) = (dnil • dnil) • const n
```

Then, there are sixteen constant in $\mathbb{D}$ that are used to indicate special meaning in the universal **WHILE** program.

```
dquote : D
dquote = const 1

d:= : D
d:= = const 2

d→→ : D
d→→ = const 3

dwhile : D
dwhile = const 4
```

```
dvar : D
dvar =  const 5

ddnil : D
ddnil = const 6

dcons : D
dcons = const 7

dhd : D
dhd = const 8

dtl : D
dtl = const 9

d=? : D
d=? = const 10

dohd : D
dohd = const 11

dotl : D
dotl = const 12

docons : D
docons = const 13

doasgn : D
doasgn = const 14

dowh : D
dowh = const 15

do=? : D
do=? = const 16
```

B.2 Code WHILE

The mapping function from the **WHILE** program to **WHILE** data is defined as:

$$\underline{\text{read } V_i;\ C;\ \text{write } V_j} \quad = \quad ((\text{vari})\underline{C}(\text{varj}))$$

$$\underline{C;\ D} \quad = \quad (;\underline{CD})$$

$\underline{\text{while E do C}}$	=	(while$\underline{\text{EC}}$)
$\underline{V_i\text{:= E}}$	=	(:=(vari)$\underline{\text{E}}$)
$\underline{V_i}$	=	(vari)
$\underline{\text{d}}$	=	(quoted)
$\underline{\text{cons E F}}$	=	(cons$\underline{\text{EF}}$)
$\underline{\text{hd E}}$	=	(hd$\underline{\text{E}}$)
$\underline{\text{tl E}}$	=	(tl$\underline{\text{E}}$)
$\underline{\text{=? E F}}$	=	(=?$\underline{\text{EF}}$)

B.3 Variable of Universal WHILE Program

We can define the eight variable of the universal **WHILE** program as:

```
PD : Fin 8
PD = zero

Pp : Fin 8
Pp = suc zero

Cc : Fin 8
Cc = suc (suc (zero))

Cd : Fin 8
Cd = suc (suc (suc (zero)))

St : Fin 8
St = suc (suc (suc (suc (zero))))

V1 : Fin 8
V1 = suc (suc (suc (suc (suc (zero)))))

W : Fin 8
W = suc (suc (suc (suc (suc (suc (zero))))))

Z : Fin 8
Z = suc (suc (suc (suc (suc (suc (suc (zero)))))))
```

B.4 Universal WHILE Program

The universal **WHILE** program which could simulate the **WHILE** program of 1 variable is defined as:

```
read PD;              (* Input (p.d) *)
  P := hd PD;         (* P = ((var 1) C (var 1)) *)
  C := hd (tl P)      (* C = hd tl p program code is C *)
  Cd := cons C nil;   (* Cd = (c.nil), Code to execute is c *)
  St := nil;          (* St = nil, Stack empty *)
  Vl := tl PD;        (* Vl = d Initial value of var.*)
  while Cd do STEP;   (* do while there is code to execute *)
write Vl;
```

Where the *STEP* is the Macro that simulate the program.

B.5 STEP Macro

When we are simulate the **WHILE** program which has only one variable, we can easily define the stack machine as a Macro based on three variables: *Cd* which is the *command*, *St* which is the stack and *V1* which is the only one variable. Here we define the syntax sugar *cons** as *cons* A B C* = *cons A* (*cons B C*). Then we could rewrite [Cd, St] by:

[((quote D)·Cr), St]	$\Rightarrow$	[Cr, cons D St]
[((var 1)·Cr), St]	$\Rightarrow$	[Cr, cons V1 St]
[((hd E)·Cr), St]	$\Rightarrow$	[cons* E dohd Cr, Sr]
[(dohd·Cr), (T·Sr)]	$\Rightarrow$	[Cr, cons (hd T) St]
[((tl E)·Cr), St]	$\Rightarrow$	[cons* E dotl Cr, St]
[(dotl·Cr), (T·Sr)]	$\Rightarrow$	[Cr, cons (tl T) Sr]
[((cons E_1 E_2)·Cr), St]	$\Rightarrow$	[cons* E_1 E_2 docons Cr, St]
[(docons·Cr), (U·(T·Sr))]	$\Rightarrow$	[Cr, cons (cons T U) Sr]
[((=? E_1 E_2)·Cr), St]	$\Rightarrow$	[cons* E_1 E_2 do=? Cr, St]
[(do=?·Cr), (U·(T·Sr))]	$\Rightarrow$	[Cr, cons (=? T U) St]
[((; C_1 C_2)·Cr), St]	$\Rightarrow$	[cons* C_1 C_2 Cr, St]
[((:= (var 1) E)·Cr), St]	$\Rightarrow$	[cons* E (cons doasgn Cr), St]
[(doasgn·Cr), (W·Sr)]	$\Rightarrow$	{Cd := Cr, St := Sr; V1 := W}
[((while E C)·Cr), St]	$\Rightarrow$	[cons* E dowh (while E C) Cr, St]

$$
\begin{array}{lcl}
[(\text{dowh}\cdot((\text{while E C})\cdot\text{Cr})),\ (\text{nil}\cdot\text{Sr})] & \Rightarrow & [\text{Cr, Sr}] \\
[(\text{dowh}\cdot((\text{while E C})\cdot\text{Cr})),\ ((\text{A}\cdot\text{B})\cdot\text{Sr})] & \Rightarrow & [\text{cons}^{*}\ \text{C (while E C) Cr, Sr}] \\
[\text{nil, St}] & \Rightarrow & [\text{nil, St}]
\end{array}
$$

B.6 Constant in Expression

First of all, we define a function that would construct some value in the format of $\mathbb{E}$ based on the the data of $\mathbb{D}$:

```
dtoE : {n : ℕ} → D → E n
dtoE dnil = nil
dtoE (d₁ • d₂) = cons (dtoE d₁) (dtoE d₂)
```

Then, there are sixteen constant in $\mathbb{E}$ that are used to indicate special meaning in the universal **WHILE** program.

```
quoteE : {n : ℕ} → E n
quoteE = dtoE dquote

varE : {n : ℕ} → E n
varE = dtoE dvar

valueE : {n : ℕ} → (f : Fin n) → E n
valueE f = dtoE (dftod f)

hdE : {n : ℕ} → E n
hdE = dtoE dhd

dohdE : {n : ℕ} → E n
dohdE = dtoE dohd

tlE : {n : ℕ} → E n
tlE = dtoE dtl

dotlE : {n : ℕ} → E n
dotlE = dtoE dotl

consE : {n : ℕ} → E n
consE = dtoE dcons

doconsE : {n : ℕ} → E n
doconsE = dtoE docons
```

```
=?E : {n : ℕ} → E n
=?E = dtoE d=?

do=?E : {n : ℕ} → E n
do=?E = dtoE do=?

→→E : {n : ℕ} → E n
→→E = dtoE d→→

:=E : {n : ℕ} → E n
:=E = dtoE d:=

doasgnE : {n : ℕ} → E n
doasgnE = dtoE doasgn

whileE : {n : ℕ} → E n
whileE = dtoE dwhile

dowhE : {n : ℕ} → E n
dowhE = dtoE dowh
```

Bibliography

[1] Computability, April 2015. Page Version ID: 659987704.

[2] Halting problem, October 2015. Page Version ID: 685183096.

[3] Markov algorithm, September 2015. Page Version ID: 680734144.

[4] Recursive language, November 2015. Page Version ID: 692264552.

[5] Recursively enumerable language, August 2015. Page Version ID: 678743023.

[6] Universal Turing machine, March 2016. Page Version ID: 709735601.

[7] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM (JACM)*, 31(3):441–458, 1984.

[8] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[9] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80. ACM, 1972.

[10] B. Jack Copeland. The church-turing thesis. *Stanford encyclopedia of philosophy*, 2002.

[11] Galina Jirásková and Matús Palmovskỳ. Kleene Closure and State Complexity. *ITAT*, 2013:94–100, 2013.

[12] Neil D. Jones. *Computability and complexity: from a programming perspective.* Foundations of computing. MIT Press, Cambridge, Mass, 1997.

[13] Eugenio Moggi. *Computational lambda-calculus and monads.* Citeseer, 1988.

[14] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[15] H. Rogers. *Theory of recursive functions and effective computability.* MIT Press, Cambridge, Mass, 1st mit press pbk. ed edition, 1987.

[16] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[17] Jaap van Oosten. *Homotopy Type Theory: Univalent Foundations of Mathematics, http://homotopytypetheory. org/book, Institute for Advanced Study.* JSTOR, 2014.

[18] Ann Yasuhara. *Recursive function theory and logic.* Computer science and applied mathematics. Academic Press, New York, 1971.