The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

The Undecidability of the Halting Problem

A Formal Proof in Agda

Submitted April 2016, in partial fulfilment of
the conditions of the award of the degree:
BSc (Hons) Computer Science.

Zongzhe Yuan: psyzy3
4234822
With Supervision from Thorsten Altenkirch

School of Computer Science and Information Technology
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature ____________________________

Date _______/_______/_______

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 points in the proof of the correctness of the universal model and the proof of the *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 that was used in the project follows the definition in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* in 1997[12].
The complete code could be found in my *GitHub* or go to the next url: `https://github.com/ZongzheYuan/HaltingProblem.git`.

Abstract

The *Halting Problem*, is a famous problem in the computability theory. Alan Turing proved the undecidability of the *Halting Problem* in 1936[12]. By choosing some computational models which are *Turing-equivalents* to the *Turing Machine*[10], defining the syntax and semantics of that model and constructing the universal computational model based on the theory of *Turing-complete*, one could prove the undecidability of the *Halting Problem.* The aim of this project is constructing a proper computational model, proving the correctness of the universal model (has the property of *Turing completeness*), and finally using that model to prove the undecidability of the *Halting Problem.* In more details, the project will develop a machine checked proof of the undecidability of the *Halting Problem* using the *Agda* system.

Keywords: Agda, Turing-completeness, Universal Model, Undecidable, Machine checked proof, WHILE language

Contents

Chapter 1

Approach to the Project

1.1 Overview

Alan Turing analysed and formalised the class of all computational procedure in 1936[16] when the well-known computational model *Turing Machine* was introduced to the world. *Alan Turing* has used the model of the *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 the *Halting problem* in 1936 as well[16], which constitute the objective of this paper.
To prove the undecidability of the *Halting problem*, one must use a computational model that is *Turing-equivalent* to the *Turing Machine* in the thesis of the *Church-Turing thesis*[10]. To simplify, we choose a computational model that contains one *input*, one *output* and a *function* from *input* to *output*. The detail definition will be shown in 2.1. We choose some proper data structure D, defined in 2.1.1, to represent the data and let the *input* and *output* belongs to D. Then we could define the *function* (in 2.1.3) as a relation $\bullet \to \bullet \subseteq D \times D$ which is also the mapping from the *input* to the *output*. To prove the correctness of the relation, we must construct this computational model with concrete syntax and semantics by using some proof assistant language – the meta language (in 2.1.2 and 2.1.3).
Then we should construct the universal model following the thesis of *Turing-complete*[10] (in 2.2). That is, if we have $f \in \mathit{function}$, *input* and $\mathit{output} \in D$, 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.
In order to prove the correctness of our predefined universal computational model,

we should define the concrete code method to code the *function* into our predefined data structure D without ambiguity (in 2.2.1). The first step in our process is to construct the interpretation simulation step s in the chosen meta language and prove the relation between the *function* f and the interpretation simulation s (in 2.2.2). 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 can get $(\lfloor f \rfloor, \epsilon, input) \Rightarrow^* (\epsilon, \epsilon, output)$ where $\Rightarrow^*$ is multistep relationship of relation $\Rightarrow$ (in 2.2.2.1). From the proof given above we know that we can use 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 (in 2.2.2.2). 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 $\Rightarrow$ (in 2.2.3). 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 the *Halting Problem*, we can 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 decides the *Halting Problem*. Then we can construct a program $m \in$ Program and feed h to m which means running h inside m. Then we let h to decide whether $m(\lfloor m \rfloor)$ will halt or not, which will end up in contradiction with the definition of m. The construction and the proof could be found in 2.3. Finally we can conclude that h doesn't exists by contradiction to our premise, and there is no model that can decide the *Halting Problem*, which otherwise means that the *Halting Problem* is undecidable.

1.2 Background

1.2.1 The Church-Turing Thesis

The *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 concepts that related to this paper are the *Turing completeness* and the *Turing equivalence.*

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

Turing equivalence said that if two computational models could simulate each one by the other, then these two computational machine is called *Turing equivalence*[15].

By the *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 *Turing Machine*[10]. Though the goal of the project is to prove the undecidability of the *Halting Problem*, constructing and formalising a universal *Turing Machine* is quite complex. Thus we can choose many other notions of effective other than the *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]

These different forms that contain same efficient computing modules share several common characteristics[12].

- The procedure consists of finite size of instructions.
- The computation is carried out in a discrete stepwise fashion but not in the method of continuous methods with analogue devices.
- The computation is carried out deterministically but not in random methods.

- Though a terminating computation must not rely on infinite amount of space or time, there is no limitation set for the amount of memory storage space or time available.

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 can construct universal computational model.

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*.

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] circumstances.
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 The Halting Problem

In the area of computability theory, the *Halting Problem* indicates that a given universal computing program (the model that is *Turing-complete*[12]) may determine any other arbitrary computing program that would return the result on arbitrary input in a finite number of steps (a finite period 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 $h(\lfloor t \rfloor \cdot \mathit{input}) \equiv false$, then h determining the *Halting Problem*.
It is easily to prove that the *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 the aims to this paper is to prove the undecidable of the *Halting Problem*.

1.2.4 The WHILE language

The *WHILE* language is a language that was just the right mixture of expressive power and simplicity. It provides the strict definitions of syntax and semantics and stays in the same level with *Turing Machine* model in terms of computing effectiveness (Turing-complete)[12]. As well the data structure of *WHILE* treats the program as data object which could solve some rather complex missions. Furthermore, with the simplicity, *WHILE* language could be simply used to prove many theorems and their behaviours. By considering those several reasons, the project will focus on proving the *Halting Problem* undecidability in the model of *WHILE* language[12]. The definition of the *WHILE* language can be found in appendix A. The implementation of the *WHILE* language in *Agda* can be found in 2.1.

1.2.5 Agda

Agda , a dependent type language and an interactive proof assistant that implements the Martin-Löf type theory[17], could assist me to develop a machine checked proof and formalise the proof of the *Halting Problem*. Because the *dependent type* allow types to talk about values, the programs written by those languages 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 integrated, not crash or non-terminate. And the mathematical proofs in *Agda* are written as structurally induction format, which are recursive functions that inducing on some inductive type argument. Thus, it is equivalent to give mathematical proof by constructing some well-typed function that could finally terminate. Therefore, *Agda* can be used as a framework to formalise formal logic systems and to prove the lemma which can be proved in mathematical ways.
We can use the key word **data** in *Agda* to define some inductive types, or more generally define some inductive families. Here we define the basic type *natural number* as an example.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Then we can define the function *plus* based on the *natural number.*

```
_+_ : ℕ → ℕ → ℕ
_+_ zero b = b
_+_ (suc a) b = suc (a + b)
```

Finally we can prove the *associative law* of our defined function by doing induction on the first argument, which is an inductive type in our definition.

```
suc-ok :  {a b : ℕ} → suc a + b ≡ suc (a + b)
suc-ok {zero} = refl
suc-ok {suc a} = cong suc refl

plus-asso : {a b c : ℕ} → (a + b) + c ≡ a + (b + c)
plus-asso {zero} = refl
plus-asso {suc a}{b}{c} = (suc a + b) + c
                        =⟨ cong (λ x → x + c) (suc-ok {a}{b}) ⟩
                        suc (a + b) + c
                        =⟨ suc-ok {a + b}{c} ⟩
                        (suc ((a + b) + c))
                        =⟨ cong suc (plus-asso {a}{b}{c}) ⟩
                        suc (a + (b + c))
                        ■
```

Using *Agda* as the proof assistant language to this project has a lot of advantages. Another convenient virtue is that we can use Unicode characters while we write program in *Agda* , which let the proofs in *Agda* look similar to those logic proofs on paper and on textbooks.
The concept and those basic technique of *Agda* could be found in the book *Dependently Typed Programming in Agda* by *Ulf Norell and James Chapman* in 2009[14].

1.3 Related Work

1.3.1 Prove the Undecidability of the Halting Problem

In 1936, *Alan Turing* has proved that a general algorithm to solve the *Halting Problem* for all possible program-input pairs cannot exist[2].
To prove the undecidability of the *Halting Problem*, we firstly assume that there exists some computational models that could decide the *Halting Problem*, then we can found some contradiction which prove that there is no computational model that could decide the *Halting Problem*, which means the *Halting Problem* is undecidable.

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 the halting set $H = \{(p, inp)|$ program p halts when it runs on input $inp\}$. Then we could prove that the following function h is not computable which means there is no total computable function that may decide whether an arbitrary program p halts on arbitrary input inp or not[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. Lastly, 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, but u will loop forever by the definition of *Turing Machine* u. Thus, u can't halt on its binary code. However in that situation h will return 0 which means u will halt finally. In conclusion, there is no universal *Turing Machine* that could decide the *Halting Problem* which can be proved by finding the contradiction to the assumption[2].

Chapter 2

Design and Implementation

2.1 The WHILE Language in *Agda*

The definition of the *WHILE* language mentioned in this section follows the definition in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* in 1997[12].

2.1.1 Tree Data Structure

The language *WHILE* that computes with a *trees* data structure is built from a finite set. Thus we define a tree data structure $\mathbb{D}$ with several related functions in *Agda* at first. To achieve it, we must define the *atoms* for the *trees* beforehand. *Atoms* mean small substances that can't be divided further into subparts. However, the complexity in defining a large number of *atoms* may make our proof become more complicated. In fact, we can define only one *atom* called *nil*, because any other values, or say other "atoms" that we presume 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 $\mathbb{D}$.

```
dfst : D → D
dfst dnil = dnil
dfst (d₁ • d₂) = d₁
```

```
dsnd : D → D
dsnd dnil = dnil
dsnd (d₁ • d₂) = d₂
```

2.1.2 Syntax

The syntax of *WHILE* is defined following the syntax definition in the book[12], and is consisted of *expression*, *command* and *program*.

2.1.2.1 Expression

The *expression* is constructed in a binary tree format, which has the same format of the data structure we defined previously. Then we can define the *expression* as 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 *expressions*, or the equality of two *expressions*. 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 contains **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' names (**zero** is the first variable and so on).

2.1.2.2 Command

A *command* is either the assignment from some *expressions* to some variables, 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 the *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 $s(x_i) = v_i$, 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 variables and their correspondence values, the program is impossible to meet the condition that one variable hasn't been defined.

2.1.3.2 The Semantics of Expression

Then we can define the evaluation function ε with the type of $\mathbb{E} \longrightarrow (\mathit{Store}^{p} \to \mathbb{D})$, which means for $e \in \mathbb{E}$ and a given *store* of program $\mathbb{P} : \sigma \in \mathit{Store}^{\mathbb{P}}$, $\varepsilon[\![e]\!]\sigma = d \in \mathbb{D}$. The definition of the evaluation function could be found 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 The Semantics of Command

The execution of a *command* in the program $\mathbb{P}$ could be used as a function $f : \mathbb{C} \times \mathit{Store}^{\mathbb{P}} \times \mathit{Store}^{\mathbb{P}}$. However, because we can't guarantee that the execution of a *command* will eventually halt and yield some output, the execution function should be a partial function $f : \mathbb{C} \times \mathit{Store}^{\mathbb{P}} \times \mathit{Store}^{\mathbb{P}}_{\perp}$. 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 $c \vdash \sigma \to \sigma' \subseteq \mathbb{C} \times \mathit{Store}^{\mathbb{P}} \times \mathit{Store}^{\mathbb{P}}$ where σ' is the new *environment* updated by the execution of *command c*. The definition of the execution *command* relationship could be found 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 The 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 *Neil D. Jones* in the paper[12], we could know that the semantics of *WHILE* is:
$[\![\bullet]\!]^{\text{WHILE}} : \mathbb{P} \to (\mathbb{D} \to \mathbb{D}_{\perp})$ defined for p = read X; C; write Y by:
$[\![\text{p}]\!]^{\text{WHILE}} = \text{e}$ if $\text{C} \vdash \sigma_0^p(\text{d}) \to \sigma$ and $\sigma(\text{Y}) = \text{e}$
If there is no e such that $[\![\text{p}]\!] = \text{e}$, then p **loops** on d; otherwise p terminates on d.
Following the definition, we define the partial relationship in *Agda* as following:

```
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 found 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 that we can't guarantee the eventual halt of the *command* or the *program* 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. No *command* could be run in *zero* step. Then the function will do induction on the *command*. The assignment step will only take *1* step. The steps that need to take relies on the sequence of two *command* c_1 and c_2 will be the sum of steps that take on c_1 and the steps that costed on c_2. Similarly in the *while* loop, the *command* will take *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 (Σ D (ExecP p inp)))
```

2.2 The Universal WHILE model

2.2.1 Interpretation of the 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 ambiguity.
Initially we should define some constants to indicate some distinct elements of $\mathbb{D}$. Those constants would represent the program in the form of $\mathbb{D}$. The definition could be found in B.1.
Then we defined the function that map the *program* to $\mathbb{D}$: *code* $\in \mathbb{P} \times \mathbb{D}$. The *code* function consists of three parts.

2.2.1.1 Code the Expression

Initially we should define the function that code the *expression* into 𝔻. The definition of the mapping function of *expression* could be found 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 found 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 the later version, I also add the number of variables of the program to the result of coding in order to allow the universal *WHILE* program could interpret any *WHILE* program with arbitrary variables in 3.3.1. The definition of the mapping function of *program* could be found 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) = (dvar • dftod x) •
                         (codeC c •
                         (dvar • dftod y))
```

2.2.1.4 Decode

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

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

2.2.2 The 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.
Then 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 Interpretation in Agda

We can imitate the simulation step following the definition using *Agda* at first. The definition of the *STEP* Macro could be found 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 $((codeC\ C \cdot Cr), St, d_1) \Rightarrow^* (Cr, St, d_2)$.

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

2.2.2.2 Interpretation in the universal WHILE program

Then we should define the 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 held 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 ⇒ and the execution of *while* loop does the induction on one step, we can prove that the several steps relation in *Agda* simulation have correspondence with 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* have the ability of simulating some *WHILE* program, then the universal *WHILE* program can simulate the same *WHILE* program.

2.2.3 Correctness of the Universal WHILE model

Finally by using the proof that were mentioned above, 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 the Halting Problem

2.3.1 Construct the WHILE program U

To prove the undecidability of the *Halting Problem*, we should construct a special *WHILE* program at first. Following the definition on *wiki*[2] and on paper[7], we could construct the program **U** defined as following. The strategy to construct **U** is mentioned in 1.3.1. However, by considering the *syntax* and *semantic* of *WHILE* program and the 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 a program **U**, and assume that there is a program h that could decide the *Halting Problem*, then the argument to the program **U** should always 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 to assign the same value to its original variable, to indicate the empty *command*.

2.3.1.1 Property 1 of U

From our 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 the program **U** and execute the program **U** on $(\lfloor h \rfloor \bullet input)$, the program **U** will never terminate.
Initially we can prove that the infinite loop can't terminate, and if there is *command* in the form of *while true command*, then this *while* loop 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 that the *assignemnt* is the leaf of the tree, *sequence* and *while* loop both have two branches.

Then we can prove that for any $h \in \mathbb{P}$, if $h(\mathit{input} \bullet (\lfloor h \rfloor \bullet \mathit{input}))$ yields *true*, then the execution of **U** on $(\lfloor h \rfloor \bullet \mathit{input})$ 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 our 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 the program **U** and execute the program **U** on $(\lfloor h \rfloor \bullet \mathit{input})$, the 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 the 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* which has the property that for all $p \in \mathbb{P}$ and $\mathit{input} \in \mathbb{D}$, if p ***halts*** on *inp*, then $h\ (\lfloor p \rfloor \bullet \mathit{input}))$ yields *true*.

```
prop₁ : ∀ {n : ℕ} → ∀ {p : P n} → ∀ {inp : D}
      → (Σ 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 that for all $p \in \mathbb{P}$ and $\mathit{input} \in \mathbb{D}$, if p doesn't ***halt*** on *inp*, then $h\ (\lfloor p \rfloor \bullet \mathit{input}))$ yields *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 the program **U** and the program h. We can name the property "**U** *halt* on $(\lfloor h \rfloor \bullet input)$" as X, "h $(\lfloor p \rfloor \bullet input))$ yielding *true*" as Y and "h $(\lfloor p \rfloor \bullet input))$ 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 *false* 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 the *Halting Problem*, which means we assume there exists a program h which could *decide* the *Halting Problem* and get $\perp$ from our assumption.

```
halt-contradiction : {h : P 1}
                   → (∀ {n : ℕ} → ∀ {p : P n} → ∀ {inp : D}
                      → (Σ 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 : Σ 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-⊥ : Σ 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 Further Work

3.1 Evaluation and Summary

This dissertation implements a formalisation of the *Turing-complete* computational model *WHILE* program, and the proof of the undecidability of the *Halting Problem* in *Agda*. It also helps readers to understand the *Church-Turing Thesis*, the *Turing Machine*, the universal computational model. Above all, it presents an accessible proof of the undecidability of the famous problem – the *Halting Problem*, and explains the details on how we can assume a hypothetical model and derive *false* from our assumption.

The idea of the *Turing-complete* model – *WHILE* language follows *Neil D. Jones*'s work in the paper *Computability and Complexity: from a Programming Perspective*[12], including a lot of personal effort and intelligence on the definition of the *WHILE* model in *Agda* and the construction of the universal *WHILE* program in *Agda*. The idea of the correctness of the universal *WHILE* model is obvious because the *WHILE* language satisfies the *Turing-complete* property. However, to prove the correctness of the universal *WHILE* model is extremely hard. The core part of the project is to find the correspondence between the execution of the *WHILE* program on some input and the universal *WHILE* program which takes the previous program as part of the argument. With the help of Dr Thorsten Altenkirch, the supervisor of this project, I developed an inspired idea which constructed the intermediate *Agda* interpretation step. Based on the intermediate *Agda* interpretation step, I constructed the relationship between the execution of the *WHILE* program and the universal *WHILE* program. Additionally, the idea of the proof of the undecidability of the

Halting Problem follows some of the *Alan Turing*'s work[2]. I also develop my own version of the proving process and the details can be found in 2.3. Since our constructed *WHILE* program *U* must take the code of some other programs as the input, the argument to the hypothetical *WHILE* program *h* (the program presumable can decide the *Halting Problem*) can't just follow the proof on paper[2]. Furthermore, I also abstract the proof on propositional level which can be found in 2.3.2.3.
There are two aspects that using *Agda* to do this project. One is to develop a machine checked proof of the undecidability of the *Halting Problem.* The *Halting Problem* is such a famous problem that almost all the people who worked around with computer science are familiar with. However, people can only find the idea of the proof of the undecidability of the *Halting Problem* theoretically. There is no strict proof using *Agda*, the interactive proof assistant language, which should formalise a computational model is *Turing-complete* and use that model to derive the problem. That is, proving the undecidability of the *Halting Problem* on paper theoretically is easy by following the idea of the *Alan Turing*'s work[2], but deriving the machine checked proof of the same problem is considerably difficult. The understanding of the method to construct the *Turing-complete* computational model and prove the correctness of that model, and the construction of the machine checked proof of the undecidability of the *Halting Problem* will help people know the details of the *Church-Turing Thesis*, the computational model and the *Halting Problem* distinctly instead of wandering at an abstract and theoretical level. The second aspects is that using *Agda* as the interactive proof assistant language has a lot of advantages. Since *Agda* allows us to use Unicode characters while we write programs in *Agda* and that may lead to the proofs in *Agda* look similar to those logic proofs on paper and textbooks. Thus, readers can go through the whole proof plainly. Combining with the previous aspect, people can read each line of the proof in depth as well as understand the idea of the construction of the machine checked proof, the general as well as the detail idea of the proof. Besides, *Agda* can automatically infer the type of the implicit argument (although sometimes which will cause the compilation of the program a little bit slow), pattern match on the inductive argument conveniently and show the type of each variable to the current goal. Furthermore, *Agda* can compute the type of the goal based on the definition of the function and current procedure of the program, check whether the provided 'answer' matches the type to the goal or not, and update the type to the consequence goals if some goals have been completed. Furthermore, lots of convenient built-in functions and the standard *Agda* library help me a lot, and make the whole machine checked proof to the problem much easier.

The management to the project follows the plan proposed at the beginning of this academic year. I used *GitHub* as the program revision control and source code management system to manage the project. I determined the object to the project at the end of last summer vacation after I went through *Neil D. Jones*'s paper and the *Agda* tutorial[12][14]. I started the project earlier than almost all the other people, and finished the construction of the *WHILE* language and the universal *WHILE* model in *Agda* before Christmas. Then I struggled with the proof of the correctness of the universal *WHILE* model for two months. At the beginning of this semester, I worked out a possible solution to the previous problem, and rushed the project along with the final proof of the undecidability of the *Halting Problem* in one month under Ambrus's help. I finished the code to the project at the beginning of March, and uploaded the final work to *GitHub*. Then I spent three weeks working out the final report of the project. Along my way, I insulted and made formal visits to my supervisor Altenkirch Thorsten once a week formally, and always talked to Ambrus Kaposi and Gabe Dijkstra to work out solution to the project all the time. Without their help, I can not finish the project on time. I learnt a lot from the procedure of the project, including the knowledge and technique to *Agda*, the knowledge in the area of the computability. Furthermore, I also figure down the way of managing a project, and manage my time to the project. During the process, my communication and writing skills have been improved a lot. Above all, this project means a lot to me and becomes one of the most significant parts of my undergraduate studys.

3.2 Main Difficulties

Since the main object – the *Halting Problem* to this project is a famous theoretical topic that is relevant to the computational theorem in computer science, and in our knowledge this project is the project that first time successfully formalised a *Turing-complete* computational model, and proved the undecidability of the *Halting Problem* in *Agda* , this project contains many difficulties and uncertainties.
Initially, choosing a proper computational model will determine the difficulty of the whole project. Constructing a *Turing Machine* in *Agda* is still hard even with the help of *Agda* as the interactive proof assistant language. I had also considered the *lambda-calculus* as the computational model in the proof. Finally I choose the *WHILE* language and the result shows that this was a wise decision.
Then proving the correctness of the universal *WHILE* program is the core of the

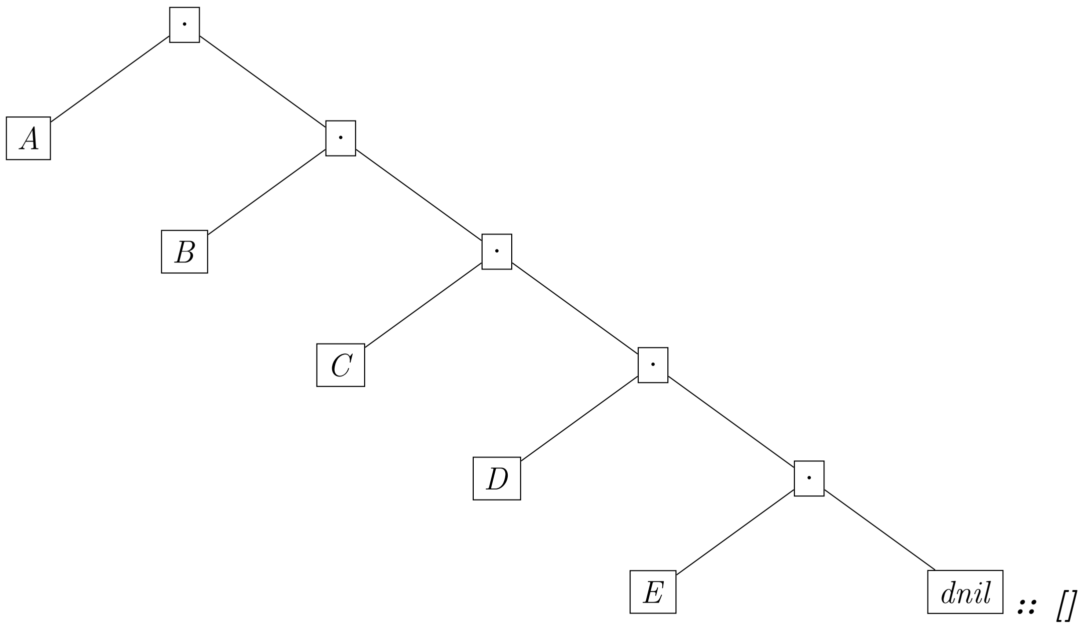
project, and it is thoroughly hard. As I said in 3.1 I found the intermediate step between the *WHILE* program and the universal *WHILE* model. In addition, even with the intermediate step, finding the relationships between *WHILE* and the intermediate step, and the relationships between the intermediate step and the universal *WHILE* model are still difficult. The execution of the *command* is a partial function and is implemented as a relationship in *Agda*. The interpretation of the *WHILE* program in *Agda* will violate the termination checker because of the possible situation of the infinite *while* loop. That is the reason I do induction on the *call tree* in the proof. Additionally, the one step relation between the intermediate step and the universal *WHILE* model is the most difficult part in the project. Initially I used a lot of implicit arguments, which caused *Agda* crashing on the calculation of the type inference. I used about thirty separate files, each file contains a helper function, including almost 4000 lines of code to finish that proof, which troubled me for two months. The final proof of the correctness of the universal *WHILE* model was too hard to prove at once, so I separated it into several steps which only makes it little bit easier.
Finally, the proof of the undecidability of the *Halting Problem* is really difficult. Since *Agda* will automatically compute the type of the goal, and expand the definition it currently used, it will cost hours of time to check the type of the goal. Without the abstraction to the final proof, I can hardly know the target of the final proof. For those reasons, the final proof is derived without auto goal type checking and auto goal type inference, and come down to a lot of basic logical rules, which makes it the one of the most difficult part in this machine checked proof.

3.3 Further Work

3.3.1 WHILE Program Variables Transformation

The universal *WHILE* program defined in 2.2 can only simulate the *WHILE* program that has only one variable. Thus the final proof can only be used in a condition when machine *h* has only one variable. Under certain circumstances, it then leads to the decision that the *Halting Problem* is *undecidable*. Additionally, the *WHILE* program that has only one variable (we name it as **WHILE-I** program) has the same computation ability compared to the *WHILE* program that has many variables. That means, the number of variables to the *WHILE* program does not matter actually, and doesn't violate the property of *Turing-completeness* to the *WHILE* program. However, since *Agda* has strict type, the program must know the number of its variables

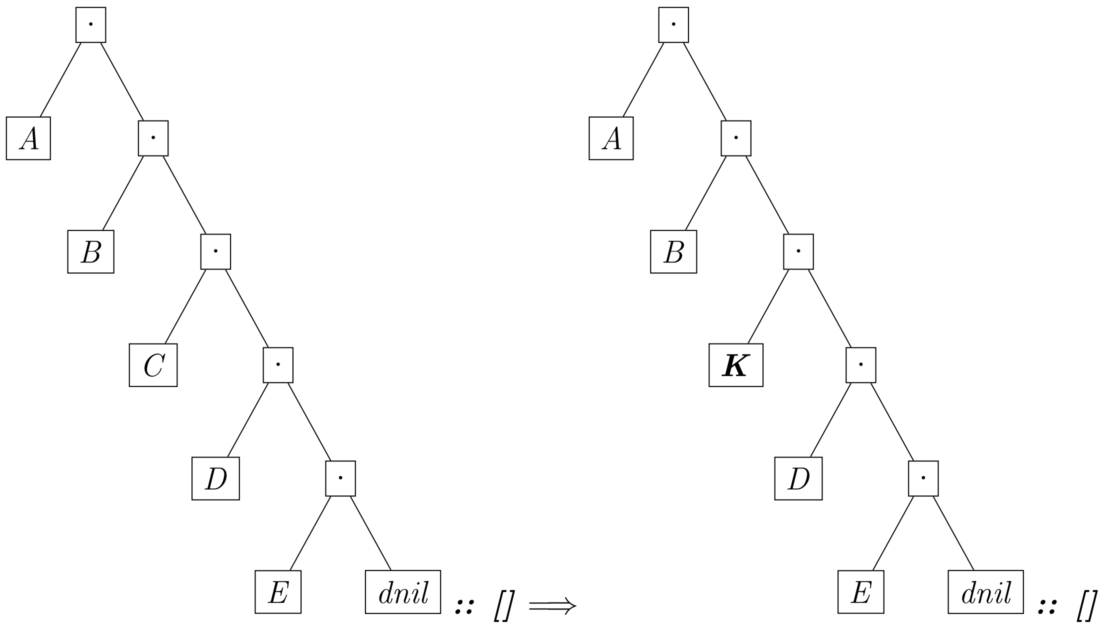
before it was defined. Thus, we can construct some rules to transform the *WHILE* program to the **WHILE-I** program and prove that they have the same effect, which means that for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, $p(input)$ yielding *output* implies that $\exists$ *p-I* $\in \mathbb{P}\ 1$, *p-I*$(input)$ yielding *output*.
Initially we should transform the *environment* of the program from **Vec D n** to **Vec D 1**, which means accumulating all the variables in the first *environment* to the first element in the second *environment* by the operator $\cdot$. For example, initially we have an *environment* of five variables: $(A :: B :: C :: D :: E ::[])$, 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 as 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 the *assignment*, for example if we assign variable C with value $K \in \mathbb{E}$, the *environment* with five variables would update as:

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

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 achieved in the future.

3.3.2 Interpretation of the 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 can conclude that our universal *WHILE* program defined in 2.2 can simulate the *WHILE* program with 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 $p(\textit{input})$, then we should transform p to $p\text{-}I$ which has only one variable by our predefined transformation function. And we know that $p(\textit{input})$ yielding *output* implies that $p\text{-}I(\textit{input})$ 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 $p\text{-}I(\textit{input})$ yielding *output* implies that $u(\lfloor p\text{-}I \rfloor \cdot \textit{input})$ yielding *output*. Thus we can conclude that for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, $p(\textit{input})$ yielding *output* implies $u(\lfloor p\text{-}I \rfloor \cdot \textit{input})$ yielding *output* which means universal *WHILE* program defined in 2.2 can simulate the *WHILE* program has arbitrary variables.
Thus for the proof of *Halting Problem*, we can say that for $p \in \mathbb{P}\ n$, h can decide the *Halting Problem* implies *false*.

Appendix A

Definition of the WHILE Language

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

A.1 Syntax of the Expression

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

A.2 Syntax of the Command

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

A.3 Semantics of the Expression

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

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

A.4 Semantics of the 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*.

$\mathrm{X{:=}E} \vdash \sigma \to \sigma[\mathrm{X} \mapsto \mathrm{d}]$ if $\varepsilon\llbracket E\rrbracket\sigma = \mathrm{d}$

$\mathrm{C;D} \vdash \sigma \to \sigma''$ if $\mathrm{C} \vdash \sigma \to \sigma'$ and $\mathrm{D} \vdash \sigma' \to \sigma''$

while E **do** C $\vdash \sigma \to \sigma''$ if C $\varepsilon\llbracket E\rrbracket\sigma \neq \mathrm{nil}$, C $\vdash \sigma \to \sigma'$,
while E **do** C $\vdash \sigma' \to \sigma''$

while E **do** C $\vdash \sigma \to \sigma$ if C $\varepsilon\llbracket E\rrbracket\sigma = \mathrm{nil}$

A.5 Example of the WHILE program

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

A.5.1 Example of the WHILE program in the WHILE model

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 of the WHILE program in *Agda*

Here we construct the same program using the definition we defined in *Agda* , it should be in 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 execute the program, we define three lists (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 :: []) :: []}
        {env₃ = list1 • list2 :: list3 :: dnil :: dnil :: []}
 tt
(seq assign assign)
(whilet {env₁ = list1 • list2 :: const 3 • list2 ::
                dnil :: ltod (2 :: 1 :: []) :: []}
        {env₂ = list1 • list2 :: const 2 • (const 3 • list2) ::
                dnil :: ltod (1 :: []) :: []}
        {env₃ = list1 • list2 :: list3 :: dnil :: dnil :: []}
 tt
(seq assign assign)
(whilet {env₁ = list1 • list2 :: const 2 • (const 3 • list2) ::
                dnil :: ltod (1 :: []) :: []}
        {env₂ = list1 • list2 :: list3 :: dnil :: dnil :: []}
        {env₃ = 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 of the Universal WHILE model

The definition of the universal *WHILE* language used in the project follows the definition in the paper *Computability and complexity: from a programming perspective* by *Neil D. Jones* 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 constants 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 the WHILE program

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

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

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

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

B.3 Variable of the Universal WHILE Program

We can define the eight variables 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 simulates the program.

B.5 STEP Macro

When we are simulating a *WHILE* program which has only one variable, we can easily define a 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]	⇒	[Cr, cons D St]
[((var 1)·Cr), St]	⇒	[Cr, cons V1 St]
[((hd E)·Cr), St]	⇒	[cons* E dohd Cr, Sr]
[(dohd·Cr), (T·Sr)]	⇒	[Cr, cons (hd T) St]
[((tl E)·Cr), St]	⇒	[cons* E dotl Cr, St]
[(dotl·Cr), (T·Sr)]	⇒	[Cr, cons (tl T) Sr]
[((cons E_1 E_2)·Cr), St]	⇒	[cons* E_1 E_2 docons Cr, St]
[(docons·Cr), (U·(T·Sr))]	⇒	[Cr, cons (cons T U) Sr]
[((=? E_1 E_2)·Cr), St]	⇒	[cons* E_1 E_2 do=? Cr, St]
[(do=?·Cr), (U·(T·Sr))]	⇒	[Cr, cons (=? T U) St]
[((; C_1 C_2)·Cr), St]	⇒	[cons* C_1 C_2 Cr, St]
[((:= (var 1) E)·Cr), St]	⇒	[cons* E doasgn Cr, St]
[(doasgn·Cr), (W·Sr)]	⇒	{Cd := Cr, St := Sr; V1 := W}
[((while E C)·Cr), St]	⇒	[cons* E dowh (while E C) Cr, St]

[(dowh·((while E C)·Cr)), (nil·Sr)]	⇒	[Cr, Sr]
[(dowh·((while E C)·Cr)), ((A·B)·Sr)]	⇒	[cons* C (while E C) Cr, Sr]
[nil, St]	⇒	[nil, St]

B.6 Constant in the Expression

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

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

Then, there are sixteen constant in 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.