A Formal Proof of the Undecidability of the Halting Problem

THORSTEN ALTENKIRCH, ZONGZHE YUAN and AMBRUS KAPOSI
School of Computer Science, University of Nottingham

We present here a formalisation of the proof of the undecidability of the Halting problem [?] for the WHILE language introduced by Jones [?] using the Agda system [?]. The main technical challenge is to implement and verify the self interpreter for the WHILE language. This work grew out of the final year project of the second author.

1. INTRODUCTION

Alan Turing analysed and formalised the class of all computational procedure in 1936 [Tur36] 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 [Tur36], which constitute the objective of this paper.

To prove the undecidability of the Halting problem, we must use a computational model that is Turing-equivalent to the Turing Machine in the thesis of the Church-Turing thesis [Cop02]. To simplify, we choose a computational model that contains only one input, one output and a function mapping from the input to the output. We choose some proper data structure D, which is in the form of binary tree, to represent the data of that computational model. The function in that computational model is defined as a relation $\bullet \to \bullet \subseteq \mathrm{D} \times \mathrm{D}$ that maps from the input to the output. Then the concrete syntax and semantics for that computational model should be defined properly by using some proof assistant language and we choose Agda, an interactive theorem proof assistant language and dependently typed programming language [Agd09] as the system in this proof.

Then we should construct the universal model following the thesis of Turing-completeness [Cop02], which means for $f \in$ function, input and output $\in$ D as we defined and f(input) derives output, then the universal model u has the property that $u(\lfloor f \rfloor \cdot \text{ input})$ derives output where $\lfloor f \rfloor$ is the code of function f in the form of D and $\cdot$ is the concat symbol in the data structure D.

In order to prove the correctness of our universal computational model, we should define the concrete code method to code the function into the data structure D without ambiguity. The first step in our process is to construct the interpretation step s in the chosen system and prove the relation between the function f and the interpretation s. The interpretation s is defined as a stack machine: (Command,Stack,Variable) ⇒ (Command',Stack',Variable') $\subseteq$ (D,D,D) $\times$ (D,D,D)

T. Altenkirch was supported by the Engineering and Physical Sciences Research Council (EPSRC) grant no. EPSRC grant EP/M016951/1 and USAF grant FA9550-16-1-0029.

which is a one-step relation between two triples. The next proof is that if we have $f \in$ function, input and output $\in D$ and f(input) derives output, then we can get $(\lfloor f \rfloor, \epsilon,$input$)\Rightarrow^*$ $(\epsilon, \epsilon,$output$)$ where $\Rightarrow^*$ is multistep relationship of relation $\Rightarrow$. The proof shows that the system we used here can interpret the function in our defined computational model. Then by constructing the universal model properly, we can prove that the interpretation step in the system has correspondence to the interpretation in the universal model in which for $f \in$ function, input and output $\in$ D and $(\lfloor f \rfloor, \epsilon,$input$)\Rightarrow^*$ $(\epsilon, \epsilon,$output$)$, the universal model u has the property that $u(\lfloor f \rfloor\cdot$input) derives output. The proof is inducted by the step of relation $\Rightarrow$. Finally we can conclude that the computational model and the universal model has the correspondence that for $f \in$ function, input and output $\in$ D and f(input) derives output, then for the universal model u, $u(\lfloor f \rfloor\cdot$input) derives output.

The proof of the undecidability of the Halting problem can be divided into three steps. Initially we assume that there exists a program $h \in$ Program which has some properties: $\forall p \in$ Program and $\forall$ input $\in$ Data, if p halts on input, then $h(\lfloor p \rfloor\cdot$ input) derives 1, else $h(\lfloor p \rfloor\cdot$ input) derives 0, which means h decides the Halting problem. Then we construct a program $m \in$ Program and feed h to m which means running h inside m and if the result of h is 1, m will loop forever, otherwise m will halt immediately. Then we run the program m on its binary code $\lfloor m \rfloor$ which means inside m, h will decide whether m will halt on $\lfloor m \rfloor$ or not, which will end up in contradiction with the definition of m. Finally we can conclude that h doesn't exists by contradicting to our premise, which means there is no model that can decide the Halting problem, which otherwise means that the Halting problem is undecidable.

2. BACKGROUND

2.1 The Church-Turing Thesis

The Church-Turing thesis is a hypothesis about the nature of computable functions [Chu36]. The thesis states that every effective computation can be carried out by a Turing Machine [Cop02]. Turing gave the definition of his thesis as the LCMs [logical computing machines: Turing's expression for Turing Machines] can do anything, that can be described as "rule of thumb" or "purely mechanical" (Turing 1948:7.) [Cop02].

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 can be used to interpret any single-taped Turing Machine [Rog87].

Turing equivalence said that if two computational models can interpret each one by the other, then these two computational machine is called Turing-equivalence [Rog87].

By the Church-Turing Thesis, any function that can be computed by some algorithm can be computed by a Turing Machine [Chu36], which means there are many computational models that is Turing equivalent to a Turing Machine [Cop02]. Constructing and formalising a universal Turing Machine is quite complex, thus we can choose many other notions of effective (Turing-complete) computational model other than the Turing Machine:

—Recursive functions as defined by Kleene [Yas71]

—The lambda calculus approach to function definitions due to Church [Mog88]

—Random access machines [CR72]

—Markov algorithms [mar15]

Those computational models listed above have been proved to be Turing-equivalence to the Turing Machine and also have the property of Turing completeness.
Universal Turing Machine is a Turing Machine that can interpret an arbitrary Turing Machine on arbitrary input [uni16]. That is for $t \in \mathrm{TM}$, input and output $\in \Sigma^*$ (Σ is the alphabet), and t(input) yields output, then the universal Turing Machine $u \in \mathrm{UTM}$ has the property that $u(\lfloor t \rfloor \cdot$ input) yields output where $\lfloor t \rfloor$ is the code of Turing Machine t.

2.2 Decidable and Non-Decidable

In the area of computability, a set S is Recursive (Decidable) $\iff$ given a set D and S $\subseteq$ D, there is a function f applies to the element $a \in$ D, f will return "true" if $a \in$ S and f will return "false" if $a \notin$ S [Jon97]. Decidable set is closed under union, intersection, complement difference and Kleene star [rec15a] circumstances. A set S is Recursively Enumerable (Semi-Decidable) $\iff$ given a set D and S $\subseteq$ D, there is a function f applies to the element $a \in$ D, f will return "true" if $a \in$ S and f will return "false" or never terminate if $a \notin$ S which means no guarantee to terminate under the element $a \notin$ S [Jon97]. If a set S is recursively enumerable and the complement of S is also recursively enumerable, then set S is Recursive [rec15b].

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-completeness [Jon97]) 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 [hal15]. That is, if there exists $h \in \mathrm{TM}$ such that $\forall t \in \mathrm{TM}$, input $\in \Sigma^*$, if for $halt_t$(input), $h(\lfloor t \rfloor \cdot$ input) derives true and for $\neg halt_t$(input), $h(\lfloor t \rfloor \cdot$ input) derives false, then h determining the Halting problem.
It is easily to prove that the Halting problem is semi-decidable [hal15] because $\forall t \in \mathrm{TM}$, input $\in \Sigma^*$, if eventually $halt_t(input)$ 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.

2.4 The WHILE language

The WHILE language is a language that was just the right mixture of expressive power and simplicity, which provides the strict definitions of syntax and semantics and stays in the same level with Turing Machine model in terms of computing effectiveness (Turing-completeness) [Jon97]. In addition, the data structure of WHILE treats the program as data object which can solve some rather complex missions that with the simplicity WHILE language can be simply used to prove many theorems and their behaviours [Jon97]. By considering those several reasons, the project will focus on proving the undecidability of the Halting Problem in the model of WHILE language. Thus, 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 [Jon97].

2.5 Agda

Agda, a dependent type language and an interactive proof assistant that implements the Martin-Löf type theory [vO14], which can assist 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 can 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. In addition, the mathematical proofs in Agda are written as structurally induction format, which are recursive functions that inducing on some inductive type argument which is equivalent to give mathematical proof by constructing some well-typed function that can 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 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 can be found in the book *Dependently Typed Programming in Agda* by *Ulf Norell and James Chapman* in 2009 [Nor09] and on the website `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php` [Agd09].

3. THE WHILE LANGUAGE IN AGDA

Part of the definition, including the definition of WHILE language that was used here follows the definition in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* in 1997 [Jon97].

3.1 Binary Tree Data Structure

The language WHILE that computes with a binary tree data structure is built from a finite set. Thus we define that binary tree data structure $\mathbb{D}$ with several related functions in Agda at first. Initially, we should 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* and because any other "atoms" that we presume to define can 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
```

3.2 Syntax

The WHILE language is consist of three level definitions, the *expression*, the *command* and the *program*, and the definition of the *expression*, the *command* and the *program* can be found in page *32* in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* in 1997 [Jon97].

The *expression* is constructed in the form of binary tree, which has the same format of the data structure we defined previously. A *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 *expressions*, or the equality of two *expressions*:

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

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 $\underbrace{\textit{suc (suc} \ \ldots \ \textit{suc (zero))}}_{n\text{ - }1}$. Then we can directly use $\underbrace{\textit{suc (suc} \ \ldots \ \textit{suc (zero))}}_{k\text{ - }1}$ to indicate the k_{th} variable's name (*zero* is the first variable and so on).

A *command* is either the assignment from some *expressions* to some variables, or the sequence of two *commands*, or the *while* loop:

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

And 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
```

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:

$$
\text{Programs} \ni \quad \mathrm{P} \quad ::= \textbf{read}\ \mathrm{X}\ \textbf{;}\ \mathrm{C}\ \textbf{;}\ \textbf{write}\ \mathrm{Y}
$$

And we 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
```

3.3 Semantics

To define the semantics of WHILE language, we must give a definition of the *Partial Function* at first [Jon97]:

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

Initially, we 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.

The definition of the semantics of WHILE language can be found in page *40* in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* in 1997 [Jon97].

Then we define the evaluation function ε with the type of $\mathbb{E} \longrightarrow (Store^p \rightarrow \mathbb{D})$, which means for $e \in \mathbb{E}$ and a given *store* of program $\mathbb{P} : \sigma \in Store^{\mathbb{P}}$, $\varepsilon[\![e]\!]\sigma = d \in \mathbb{D}$. Following the definition given by *Neil D. Jones* in the paper[Jon97], the semantics of the evaluation function ε in WHILE should be: for $e \in \mathbb{E}$ and a given *store* of program $\mathbb{P}$, $\sigma \in Store^{\mathbb{P}}$ $\varepsilon[\![e]\!]\sigma = d \in \mathbb{D}$.

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

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
```

The execution of a *command* in the program $\mathbb{P}$ can 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 , following the definition given by *Neil D. Jones* in the paper [Jon97]

$\mathrm{X{:=}E} \vdash \sigma \to \sigma[\mathrm{X} \mapsto \mathrm{d}]$ if $\varepsilon[\![E]\!]\sigma = \mathrm{d}$
$\mathrm{C;D} \vdash \sigma \to \sigma''$ if $\mathrm{C} \vdash \sigma \to \sigma'$ and $\mathrm{D} \vdash \sigma' \to \sigma''$
while E **do** $\mathrm{C} \vdash \sigma \to \sigma''$ if $\mathrm{C}\ \varepsilon[\![E]\!]\sigma \neq \mathrm{nil}$, $\mathrm{C} \vdash \sigma \to \sigma'$,
while E **do** $\mathrm{C} \vdash \sigma' \to \sigma''$
while E **do** $\mathrm{C} \vdash \sigma \to \sigma$ if $\mathrm{C}\ \varepsilon[\![E]\!]\sigma = \mathrm{nil}$

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
```

Similar to the definition of execution of $\mathbb{C}$ in WHILE, the execution of program $\mathbb{P}$ should also be defined as a relationship:

$[\![\bullet]\!]^{\text{WHILE}} : \mathbb{P} \to (\mathbb{D} \to \mathbb{D}_{\perp})$ defined for p = read X; C; write Y by:
$[\![\mathrm{p}]\!]^{\text{WHILE}} = \mathrm{e}$ if $\mathrm{C} \vdash \sigma_0^p(\mathrm{d}) \to \sigma$ and $\sigma(\mathrm{Y}) = \mathrm{e}$

If there is no e such that $[\![\mathrm{p}]\!] = \mathrm{e}$, 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)
```

3.4 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 steps. 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 stpes:

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

The idea of that function is to do induction on the step number k at first. No *command* can 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 can 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 can be executed in k steps, then a program p = (read X; c; write Y) can also be executed in k steps:

```
kStepP : {n : ℕ} → (time : ℕ) → (p : P n) → (inp : D)
        → (Maybe (Σ D (ExecP p inp)))
kStepP time (prog x c y) inp with kStepC time c (updateE x inp initialVec)
kStepP time (prog x c y) inp | just rc = just (dlookup y (ResultCT.out rc) ,
                                               terminate x y (ResultCT.exe rc))
kStepP time (prog x c y) inp | nothing = nothing
```

4. THE UNIVERSAL WHILE MODEL

Part of the definition, including the definition of the universal WHILE model that was used here follows the definition in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* in 1997 [Jon97].

4.1 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 form of $\mathbb{D}$ in order to feed the program as the input to the universal WHILE program later. It is important to define the operator • of our data structure $\mathbb{D}$ with no association, which would avoid ambiguity.

Following the definition ”Let { :=, ;, while, var, quote, cons, hd, tl, =?, nil } denote 10 distinct elements of $\mathbb{D}$, and let { dohd, dotl, docons, doasgn, dowh, do=? } denote 6 more values in $\mathbb{D}$” in page *67* in the paper *Computability and Complexity: from a Programming Perspective* by *Neil D. Jones* [Jon97], we define some constants to indicate some distinct elements of $\mathbb{D}$, which would represent the *program* in the form of $\mathbb{D}$.

Then we defined the function that map the *program* to $\mathbb{D}$: $code \in \mathbb{P} \times \mathbb{D}$. The mapping function from the WHILE program to WHILE data is defined as:

$$\underline{\text{read V}_i\text{; C; write V}_j} \quad = \quad ((\text{vari})\underline{\text{C}}(\text{varj}))$$

$$\begin{array}{lcl}
\underline{\text{C; D}} & = & (;\underline{\text{CD}}) \\
\underline{\text{while E do C}} & = & (\text{while}\underline{\text{EC}}) \\
\underline{\text{V}_i\text{:= E}} & = & (:=(\text{vari})\underline{\text{E}}) \\
 & & \\
\underline{\text{V}_i} & = & (\text{vari}) \\
\underline{\text{d}} & = & (\text{quoted}) \\
\underline{\text{cons E F}} & = & (\text{cons}\underline{\text{EF}}) \\
\underline{\text{hd E}} & = & (\text{hd}\underline{\text{E}}) \\
\underline{\text{tl E}} & = & (\text{tl}\underline{\text{E}}) \\
\underline{\text{=? E F}} & = & (=?\underline{\text{EF}})
\end{array}$$

Initially we should define the function that code the *expression* into $\mathbb{D}$. The definition of that function in Agda following the the definition above:

```
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)
```

Then we should define the function that code the *command* into $\mathbb{D}$. The definition of that function in Agda following the the definition above:

```
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)
```

Finally we should define the function that code the *program* into $\mathbb{D}$. The definition of that function in Agda following the the definition above:

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

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 can interpret any WHILE program with arbitrary variables in the future work.

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)
decodeE = ?
```

```
decodeC : {n : ℕ} → D → Maybe (C n)
decodeC = ?

decodeP : D → Maybe (Σ ℕ P)
decodeP = ?
```

4.2 The Universal WHILE model

Initially we should define the 8 variables in the WHILE language: {PD, P, C, Cd, St, V1, W, Z}.
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 can interpret other WHILE program which has only one variable. The universal WHILE program which could interpret 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 interprets the program. 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
```

When we are interpreting 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]

Initially we 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. Then we define the interpretation step following the definition using Agda:

```
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 >
```

Furthermore, we define the multi-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 >
  ⇒*-m = ?
  ⇒*-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 >
  ⇒*-b = ?
```

Then we can prove the relationship between the WHILE language and the interpretation step on *expression* e, such that for $E \in \mathbb{E}$, $\varepsilon[\![E]\!][V1 \mapsto d] = d_e$, then $((codeE \cdot Cr), St, d) \Rightarrow^* (Cr, (d_e \cdot St), d)$, and the proof is inducted by the *expression.*

```
⇒*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 >
⇒*e = ?
```

After that, we can prove the relationship between the WHILE language and the interpretation step on execution of *command*, such that for $C \in \mathbb{C}$, $C \vdash [V1 \mapsto d_1] \Rightarrow [V1 \mapsto d_2]$, then $((codeC\ C \cdot Cr), St, d_1) \Rightarrow^* (Cr, St, d_2)$, and the proof is inducted by the *command.*

As we know the *command* can be the form of *while* loop, and when we do induction on the *while* loop *while e c*, the *command* to the subproblem will also be *while e c*. The only change to the subproblem is the *environment*. However, those will violate the *termination checker* in Agda, which causes the *termination checking fail* error. In Agda, when we do induction on some inductive variables, the variables in the subproblem should be 'smaller' than the variables in the original problem, in which way Agda will treat the function as that could be terminated eventually. Even the *command* can meet the situation of *infinite while loop*, the type to the relation is consistent, and the proof function should eventually be terminated. Thus, we need some auxiliary data type to indicate that after one step, the *command* to the subproblem is definitely smaller than the *command* to the original problem. Here we called the new data type *call tree*.

```
data callTree : Set where
  cnil : callTree
  cons : callTree → callTree → callTree
```

The *assignment* in the *command* only costs one step, which is the *cnil* in the *call tree*. The *sequence* (*seq* c_1 c_2) costs $callTree_1$ for c_1 and $callTree_2$ for c_2. The *while* loop (*while e c*) costs $callTree_1$ for c and $callTree_2$ for the consequence (*while e c*). Compare to (*cons* $callTree_1$ $callTree_2$), $callTree_2$ is smaller without doubt. Thus, the *command* to the subproblem is definitely smaller than the *command* to the original problem. And because both *callTree* and $\mathbb{D}$ has the form of binary tree, we use $\mathbb{D}$ instead of *callTree*, and the proof function should be:

```
loop-ct : {n : ℕ}{c : C n}{env₁ env₂ : Vec D n} → c ⊢ env₁ ⇒ env₂ → D
loop-ct (whilef x) = dnil
loop-ct (whilet x p p₁) = (loop-ct p) • (loop-ct p₁)
loop-ct assign = dnil
loop-ct (seq p p₁) = (loop-ct p) • (loop-ct p₁)

⇒*proof : {t : D} → (c : C 1) → (d₁ d₂ Cr St : D) → (out : Vec D 1)
       → (p : c ⊢ updateE zero d₁ initialVec ⇒ out)
       → dlookup zero out ≡ d₂
       → loop-ct p ≡ t
       → < codeC c • Cr , St , d₁ > ⇒* < Cr , St , d₂ >
⇒*proof = ?
```

In the proof, we initially do induction on *command*. For the case of *while* loop, we do induction on the '*callTree*' *t* to indicate that the execution of *while e c* in the subproblem is definitely 'smaller' than that in the original problem.

```
⇒*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 >
⇒*ok c d₁ d₂ Cr St out p q = ⇒*proof {loop-ct p} c d₁ d₂ Cr St out p q refl
```

This proof shows that the execution of the *command* in WHILE language has relationship with the relation ⇒, which means that one step of execution of *command* is corresponding to the multi-step $\Rightarrow^*$ accordingly, which is the interpretation of WHILE program in *Agda*.

Then we define the real universal WHILE program. The most important part is the *STEP* Macro. Before we defining the Macro, we should define those sixteen constants in the form of $\mathbb{E}$, and we can define *STEP* as a *command* in Agda:

```
STEP-I : C 8
STEP-I =  ?
```

We don't have pattern match in WHILE language, thus we use a bunch of *if-else* we defined previously to define the *STEP* macro in the form of *command* in WHILE language.

Then we can prove that the interpretation holds 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 :: [])
c-h d1 .d1 .((dquote • d) • Cr) Cr St .(d • St) (equote d .Cr .St .d1) = case1
c-h d1 .d1 .((dvar • dnil) • Cr) Cr St .(d1 • St) (evar1 .Cr .St .d1) = case2
c-h d1 .d1 .((dhd • E) • Cr) .(E • (dohd • Cr)) St .St (ehd E Cr .St .d1) = case3
c-h d1 .d1 .(dohd • Cr) Cr .(T • St) .(dfst T • St) (edohd T .Cr St .d1) = case4
c-h d1 .d1 .((dtl • E) • Cr) .(E • (dotl • Cr)) St .St (etl E Cr .St .d1) = case5
c-h d1 .d1 .(dotl • Cr) Cr .(T • St) .(dsnd T • St) (edotl T .Cr St .d1) = case6
c-h d1 .d1 .((dcons • (E1 • E2)) • Cr) .(E1 • (E2 • (docons • Cr)))
                                        St .St (econs E1 E2 Cr .St .d1) = case7
c-h d1 .d1 .(docons • Cr) Cr .(U • (T • St)) .((T • U) • St)
                                                (edocons U T .Cr St .d1) = case8
c-h d1 .d1 .((d=? • (E1 • E2)) • Cr) .(E1 • (E2 • (do=? • Cr))) St .St
                                                (e=? E1 E2 Cr .St .d1) = case9
c-h d1 .d1 .(do=? • Cr) Cr .(U • (T • St)) .(_ • St)
                                                (edo=? U T .Cr St .d1) = case10
c-h d1 .d1 .((d→→ • (C1 • C2)) • Cr) .(C1 • (C2 • Cr)) St .St
                                                (e→→ C1 C2 Cr .St .d1) = case11
c-h d1 .d1 .((d:= • ((dvar • dnil) • E)) • Cr) .(E • (doasgn • Cr)) St .St
                                                        (e:= E Cr .St .d1) = case12
c-h d1  d2 .(doasgn • Cr) Cr .(d2 • St) St (edoasgn .d2 .Cr .St .d1) = case13
c-h d1 .d1 .((dwhile • (E • C1)) • Cr) .(E • (dowh • ((dwhile • (E • C1)) • Cr)))
                                        St .St (ewhile E C1 Cr .St .d1) = case14
c-h d1 .d1 .(dowh • ((dwhile • (E • C1)) • Cr)) Cr .(dnil • St) St
                                                (edowhf E C1 .Cr .St .d1) = case15
c-h d1 .d1 .(dowh • ((dwhile • (E • C1)) • Cr)) .(C1 • ((dwhile • (E • C1)) • Cr))
                        .((X • Y) • St) St (edowht E C1 X Y Cr .St .d1) = case16
c-h d1 .d1 .dnil .dnil St .St (enil .St .d1) = case17
```

We do induction on Cr_1 at first. The proof has been divided into several files because in order to make the compilation of that proof faster in Agda, we must provide the *environment* to each step of the execution of the *command STEP-I*, which will take up a lot of space.

```
case1 : {d St Cr d1 C P Pd : D}
        → STEP-I ⊢ Pd :: P :: C :: (dquote • d) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: d • St :: d1 :: dnil :: dnil :: [])
case2 : {St Cr d1 C P Pd : D}
        → STEP-I ⊢ Pd :: P :: C :: (dvar • dnil) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: d1 • St :: d1 :: dnil :: dnil :: [])
```

```
case3 : {Cr E St d1 C P Pd : D}
        → STEP-I ⊢ Pd :: P :: C :: (dhd • E) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: E • (dohd • Cr) :: St :: d1 :: dnil :: dnil :: [])
case4 : {St T Cr d1 C P Pd : D}
        →  STEP-I ⊢ Pd :: P :: C :: dohd • Cr :: T • St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: dfst T • St :: d1 :: dnil :: dnil :: [])
case5 : {Cr E St d1 C P Pd : D}
        → STEP-I ⊢ Pd :: P :: C :: (dtl • E) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: E • (dotl • Cr) :: St :: d1 :: dnil :: dnil :: [])
case6 : {St T Cr d1 C P Pd : D}
        → STEP-I ⊢ Pd :: P :: C :: dotl • Cr :: T • St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: dsnd T • St :: d1 :: dnil :: dnil :: [])
case7 : {Cr E1 E2 St d1 C P Pd : D}
        →  STEP-I ⊢ Pd :: P :: C :: (dcons • (E1 • E2)) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: E1 • (E2 • (docons • Cr)) :: St :: d1 :: dnil :: dnil :: [])
case8 : {Cr T U St d1 C P Pd : D}
        →  STEP-I ⊢ Pd :: P :: C :: docons • Cr :: U • (T • St) :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: (T • U) • St :: d1 :: dnil :: dnil :: [])
case9 : {Cr E1 E2 St d1 C P Pd : D}
        →  STEP-I ⊢ Pd :: P :: C :: (d=? • (E1 • E2)) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: E1 • (E2 • (do=? • Cr)) :: St :: d1 :: dnil :: dnil :: [])
case10 : {Cr U T ST d1 C P Pd : D}
        →  STEP-I ⊢ Pd :: P :: C :: do=? • Cr :: U • (T • ST) :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: dequal T U • ST :: d1 :: dnil :: dnil :: [])
case11 : {Pd P C C1 C2 Cr St d1 : D}
        →  STEP-I ⊢ Pd :: P :: C :: (d→→ • (C1 • C2)) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: C1 • (C2 • Cr) :: St :: d1 :: dnil :: dnil :: [])
case12 : {Pd P C E Cr St d1 : D}
        →  STEP-I ⊢ Pd :: P :: C :: (d:= • ((dvar • dnil) • E)) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: E • (doasgn • Cr) :: St :: d1 :: dnil :: dnil :: [])
case13 : {Pd P C Cr d1 d2 St : D}
        →  STEP-I ⊢ Pd :: P :: C :: doasgn • Cr :: d2 • St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: St :: d2 :: dnil :: dnil :: [])
case14 : {Pd P C E C1 Cr d1 St : D}
        →  STEP-I ⊢ Pd :: P :: C :: (dwhile • (E • C1)) • Cr :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: E • (dowh • ((dwhile • (E • C1)) • Cr)) :: St :: d1 :: dnil :: dnil :: [])
case15 : {Pd P C E C1 Cr d1 St : D}
        →  STEP-I ⊢ Pd :: P :: C :: dowh • ((dwhile • (E • C1)) • Cr) :: dnil • St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: Cr :: St :: d1 :: dnil :: dnil :: [])
case16 : {Pd P C E C1 Cr X Y d1 St : D}
        →  STEP-I ⊢ Pd :: P :: C :: dowh • ((dwhile • (E • C1)) • Cr) :: (X • Y) • St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: C1 • ((dwhile • (E • C1)) • Cr) :: St :: d1 :: dnil :: dnil :: [])
case17 : {Pd P C d1 St : D}
        →   STEP-I ⊢ Pd :: P :: C :: dnil :: St :: d1 :: dnil :: dnil :: []
                ⇒ (Pd :: P :: C :: dnil :: St :: d1 :: dnil :: dnil :: [])
```

Because both the relation ⇒ and the execution of *while* loop does the induction on one step, we can prove that the multi-steps relation in Agda interpretation have correspondence with the Macro *STEP* by doing induction on ⇒.

```
c-h* : {Pd P C : D}(d1 d2 Cr St1 St2 : D)
       → < Cr , St1 , d1 > ⇒* < dnil , St2 , d2 >
       → while (var Cd) STEP-I ⊢ (Pd :: P :: C :: Cr :: St1 :: d1 :: dnil :: dnil :: [])
                                ⇒ (Pd :: P :: C :: dnil :: St2 :: d2 :: dnil :: dnil :: [])
c-h* = ?

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 :: [])
step-I-ok c d₁ d₂ p = c-h* d₁ d₂ (codeC c • dnil) dnil dnil p
```

From this proof we can know that if Agda have the ability of interpreting some WHILE program, then the universal WHILE program can interpret the same WHILE program.

Finally by using the proofs we have done, 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
execP-uni = ?
```

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

5. PROOF TO THE HALTING PROBLEM USING ONE-VARIABLE PROGRAM

5.1 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* [hal15] and on paper [BM84], we can construct the program **U**. **U** will take the code of some other WHILE program $\lfloor v \rfloor$ as input and interpret the running of the hypothetical computation model h which could decide the Halting problem on $(\lfloor v \rfloor \bullet \lfloor v \rfloor)$. If h returns *true* which means $\lfloor v \rfloor$ halts on $\lfloor v \rfloor$, **U** will loop forever, otherwise **U** will terminate immediately. However, by considering the *syntax* and *semantic* of WHILE program and the universal WHILE program, we know that we must feed the code of the hypothetical computation model h into our special program **U** as part of the argument. That is, when we are constructing the program **U**, and assume that there is a program h that can decide the Halting problem, 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* in WHILE language, we use the command $x :=$ var x which is to assign the same value to its original variable, to indicate the empty *command.*

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* as we mentioned before, 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 $h(input \bullet (\lfloor h \rfloor \bullet input))$ yields *true*, then the execution of **U** on $(\lfloor h \rfloor \bullet input)$ will never terminate.

```
execC-U-loop : {t : D}{h : P 1} → (d₁ d₂ : D)
             → ExecP h (d₁ • ((codeP h) • d₁)) d₂
             → (d₂ ≡ dnil → ⊥)
             → ∀ {env : Vec D 8}
             → (p : (getCommandP U) ⊢ updateE PD ((codeP h) • d₁) initialVec ⇒ env)
             → loop-ct p ≡ t
             → ⊥
execC-U-loop = ?

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 → ⊥)
execP-U-loop = d₁ d₂ p₁ q (terminate .zero .(suc (suc (suc (suc (suc zero))))) x)
       = execC-U-loop d₁ d₂ p₁ q x refl
```

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 *false*, then if we feed h to the program **U** and execute the program **U** on $(\lfloor h \rfloor \bullet input)$, 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
execP-U-halt = ?
```

5.2 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 mentioned previously.

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

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

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

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

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⊥))
```

Finally we can prove the undecidability of the Halting problem following the abstract proof in a way that assuming there exists a program h which can decide the Halting problem and get $\perp$ from the 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
```

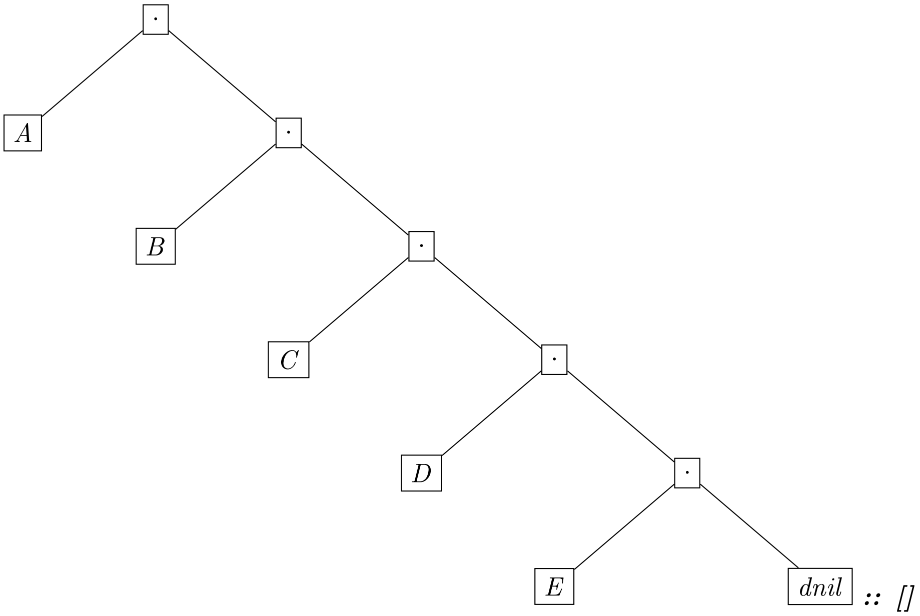
6. PROOF TO THE HALTING PROBLEM USING ARBITRARY-VARIABLE PROGRAM

6.1 WHILE Program Variables Transformation

The universal WHILE program defined previously can only simulate the WHILE program that has only one variable. Thus the the proof can only be used in a condition when machine h has only one variable and 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 $\exists\ p\text{-}I \in \mathbb{P}\ 1$, $p\text{-}I(input)$ 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 $\cdot$. In our definition, the combination operator $\cdot$ in our binary tree data structure $\mathbb{D}$ has no association. And by considering the definition of the combination operator :: in the Vector, we can also represent a Vector of data in the from of binary tree. Thus, we can transform a Vector of data in the form of $\mathbb{D}$ into single $\mathbb{D}$ data.

```
env-multi-one : {n : ℕ} → Vec D n → Vec D 1
env-multi-one [] = dnil :: []
env-multi-one (x :: v) = x • head (env-multi-one v) :: []
```

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.

```
e-multi-one : {n : ℕ} → E n → E 1
e-multi-one (var x) = hd (e-var x (var zero))
e-multi-one nil = nil
e-multi-one (cons e e1) = cons (e-multi-one e) (e-multi-one e1)
e-multi-one (hd e) = hd (e-multi-one e)
e-multi-one (tl e) = tl (e-multi-one e)
e-multi-one (e =? e1) = (e-multi-one e) =? (e-multi-one e1)
```

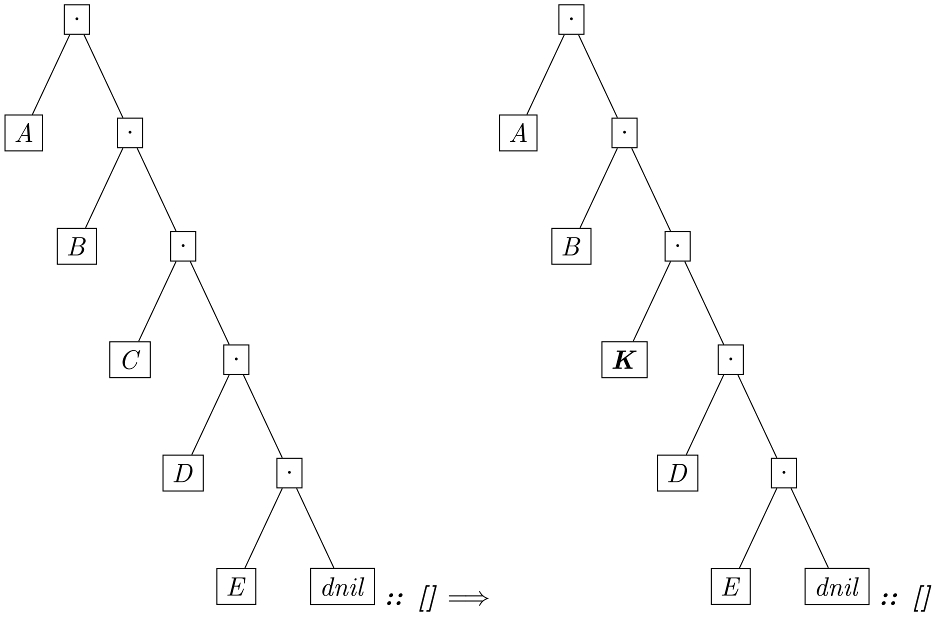
We must prove the correctness of our transformation under the *expression*. That is, $\forall e \in E, \forall env \in Env,\ \varepsilon[\![e]\!]env \equiv \varepsilon[\![transform - e]\!]transform - env$.

```
e-multi-one-ok : {n : ℕ} → (e : E n) → {env : Vec D n}
                 → eval e env ≡ eval (e-multi-one e) (env-multi-one env)
e-multi-one-ok = ?
```

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

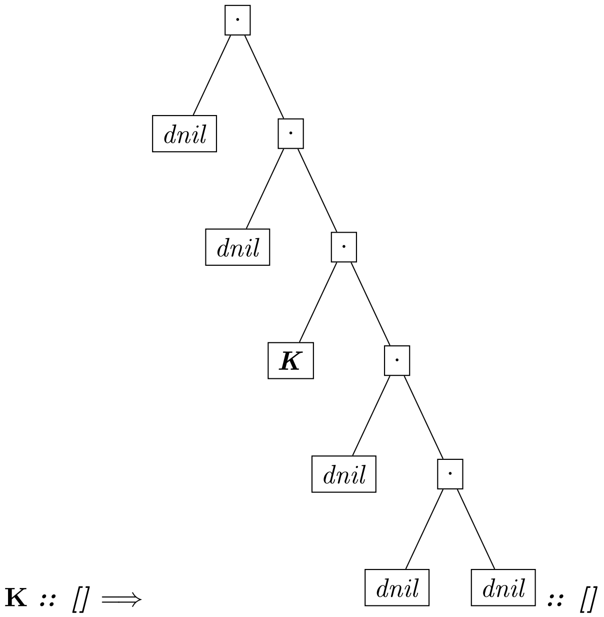


```
c-multi-one : {n : ℕ} → C n → C 1
c-multi-one (x := e) = zero := c-var x zero (e-multi-one e)
c-multi-one (c →→ c₁) = c-multi-one c →→ c-multi-one c₁
c-multi-one (while x c) = while (e-multi-one x) (c-multi-one c)
```

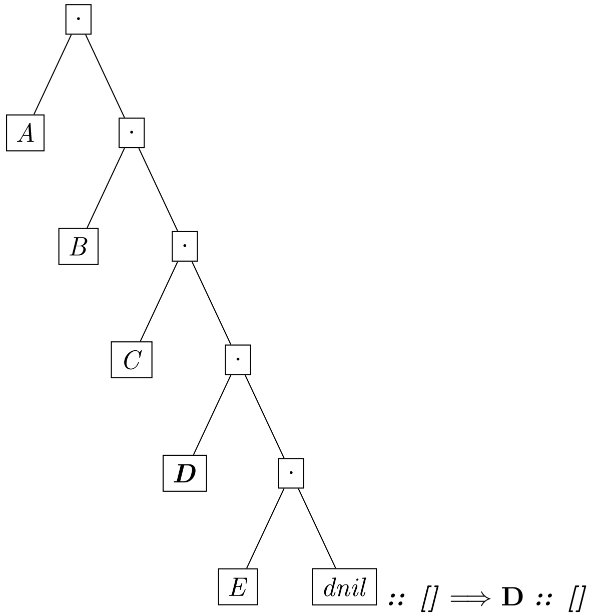
Similarly to the *expression*, we must prove our *command* transformation is correct.

```
c-multi-one-ok : {n : ℕ} → {c : C n} → {env₁ env₂ : Vec D n}
                 → c ⊢ env₁ ⇒ env₂
                 → c-multi-one c ⊢ (env-multi-one {n} env₁)
                                 ⇒ (env-multi-one {n} env₂)
c-multi-one-ok = ?
```

Finally we can transform the *program*. Because of our definition of the execution of the *program*, we must add a new *command* to construct the initial environment at first, and a new *command* to extract the target output from the environment at last. Because our transformed *program* has only one variable, thus the input will be store at variable *zero*. We must use a *command* to construct the environment from one variable to the "ideal" accumulated one variable environment. For example, the original *program*, has 5 variables, reads input **K** into the variable **C**, then in our transformation we must do the same corresponding things.



Then we can transform the *command* following our previous definition. Then we should extract the result from the accumulated one-variable environment. For example the original *program* will write the value to the variable **D** as the output, then our transformed *program* should have the same out put. However after the execution of the *command*, we will get an accumulated one-variable environment, and we will write the variable *zero* as out put. Thus, the *command* will extract the value to the variable and replace the whole variable *zero* with the output.



```
p-multi-one-h : {m : ℕ}(ini : E 1) → (inp : E 1) → (pos : Fin m) → ℕ → E 1
p-multi-one-h ini inp zero n = cons inp (e-var (fromℕ n) ini)
p-multi-one-h ini inp (suc pos) n = cons nil (p-multi-one-h ini inp pos (suc n))

p-multi-one : {n : ℕ} → P n → P 1
p-multi-one {n}(prog x c y)
= prog zero ((zero := p-multi-one-h (dtoE {1}
                                          (head (env-multi-one (initialVec {n}))))
                                    (var zero) x (suc zero))
              →→ c-multi-one c
              →→ (zero := e-multi-one (var y)))
       zero
```

We must prove our transformation of the *program* is correct. We can divide our transformed *program* into three parts: constructing the environment, the transformed *command* and extracting the output. We should prove their correctness separately at first.

```
p-lemma1 : {n : ℕ}{x : Fin n}{d : D}
         → ((zero := p-multi-one-h (dtoE {1} (head (env-multi-one (initialVec {n}))))
                     (var zero) x (suc zero))
           ⊢ d :: [] ⇒  env-multi-one (updateE x d (initialVec {n})))
p-lemma1 = ?

p-lemma2 : {n : ℕ}{y : Fin n}{env : Vec D n}
         → ((zero := e-multi-one (var y)) ⊢ env-multi-one env ⇒ (dlookup y env :: []))
p-lemma2 = ?
```

Finally we should prove the correctness of the whole *program* transformation.

```
p-multi-one-ok : {n : ℕ} → {p : P n} → {d1 d2 : D}
                 → ExecP p d1 d2
                 → ExecP (p-multi-one p) d1 d2
p-multi-one-ok = ?
```

6.2 Proof the Undecidability of the Halting Problem

As 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 can simulate the WHILE program with arbitrary variables, furthermore, we can conclude that we can prove the undecidability of the halting problem in the system we built. Here gives an example on how could we use the transformation step to execute the arbitrary WHILE program using one variable, for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, if we want to use our universal WHILE program to simulate $p(\mathit{input})$, then we should transform p to *p-I* which has only one variable by our predefined transformation function. And we know that $p(\mathit{input})$ yielding *output* implies that $p\text{-}I(\mathit{input})$ 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 $p\text{-}I(\mathit{input})$ yielding *output* implies that $u(\lfloor p\text{-}I \rfloor \cdot \mathit{input})$ yielding *output*. Thus we can conclude that for $p \in \mathbb{P}\ n$ and *input*, *output* $\in \mathbb{D}$, $p(\mathit{input})$ yielding *output* implies $u(\lfloor p\text{-}I \rfloor \cdot \mathit{input})$ yielding *output* which means universal WHILE program 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*.

```
halt-contradiction-arb : ∀ {m} {h : P m}
                   → (∀ {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-arb {m} {h} p = halt-contradiction {p-multi-one {m} h}
                                   (λ {n : ℕ} {p1 : P n} {inp : D}
                                   → (λ x → p-multi-one-ok {m} {h}
                                                           {(codeP p1) • inp}
                                                           {dtrue}
                                                           (proj1 (p {n} {p1} {inp}) x))
                                   ,'(λ x → p-multi-one-ok {m} {h}
                                                           {(codeP p1) • inp}
                                                           {dfalse}
                                                           (proj2 (p {n} {p1} {inp})
                                                                  (λ y → x y))))
```

7. CONCLUSION

In this paper, we construct a self-interpret system using Agda, following the definition by *Neil D. Jones* in 1997 [Jon97]. We prove the correctness of our system, and the property of Turing-completeness and Turing-equivalence among this system. Then we pick a kind of particular programs, the programs have one one variable, and prove the undecidability of the Halting problem using the interpretation under the program that has only one variable. Furthermore, we construct a transformation function which will transform the program that has arbitrary variables to the program that has only one variable. We prove the correctness of the transformation, which means in our system the program has many variables has the same computation power compare to some programs that has only one variable. Finally, we can use our transformation function to prove the undecidability of the Halting problem in our system.

References

[Agd09] Agda team. Agda, 2009. http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php.

[BM84] 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.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[Cop02] B. Jack Copeland. The church-turing thesis. *Stanford encyclopedia of philosophy*, 2002.

[CR72] 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.

[hal15] Halting problem, October 2015. Page Version ID: 685183096.

[Jon97] Neil D. Jones. *Computability and complexity: from a programming perspective*. Foundations of computing. MIT Press, Cambridge, Mass, 1997.

[mar15] Markov algorithm, September 2015. Page Version ID: 680734144.

[Mog88] Eugenio Moggi. *Computational lambda-calculus and monads*. Citeseer, 1988.

[Nor09] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[rec15a] Recursive language, November 2015. Page Version ID: 692264552.

[rec15b] Recursively enumerable language, August 2015. Page Version ID: 678743023.

[Rog87] H. Rogers. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, Mass, 1st mit press pbk. ed edition, 1987.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[uni16] Universal Turing machine, March 2016. Page Version ID: 709735601.

[vO14] Jaap van Oosten. *Homotopy Type Theory: Univalent Foundations of Mathematics, http://homotopytypetheory. org/book, Institute for Advanced Study*. JSTOR, 2014.

[Yas71] Ann Yasuhara. *Recursive function theory and logic*. Computer science and applied mathematics. Academic Press, New York, 1971.