Starter Questions

Feel free to discuss these with your neighbour:

- How many \texttt{while} programs compute the function $f(x) = x + 1$?
- How many Java programs are there?
- How many subsets of $\mathbb{N}$ are there?
- What is the relationship between $\mathbb{N} \rightarrow \mathbb{B}$ and $\mathbb{N} \times \mathbb{N}$?
- What is the relationship between $\mathbb{B} \rightarrow \mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$?

Reminder: Wednesday 1pm LF15, extra session on correctness proofs

Reminder: Exam removes choice this year. The semantics and correctness \texttt{while} rules will be provided.
Remaining weeks

- Wed 26, extra session on correctness proofs (LF15 1pm)
- Mon 24, Fri 28 on Computability
- Mon 1 is a bank holiday
- Fri 5 May, Part II revision/summary
- Mon 8 May, Computability examples class, Part I revision lecture

Exam

- Choice removed this year - have to do all questions
- One half of the exam = 1.5 questions from previous years
- Past papers and information about relevant questions on webpage

Handouts

- There is an additional sheet on correctness proofs online
- Chapter 4 will be available from SSO later today
Lecture 8
Computable and Uncomputable Functions
COMP11212

Giles Reger

April 2017
Reminder

So far in this course we have seen:

1. How to write programs as descriptions of computations
2. How to unambiguously describe such computations
3. How to check that a program correctly computes a function
4. How to reason abstractly about the efficiency of a program

What is left to do: **Computability**

- Define computable and uncomputable functions/predicates
- Prove the existence of uncomputable functions
- Look at the famous Halting Problem
- Consider the universality of computation
Using a while program

Universality

Coding

$f : \mathbb{N} \rightarrow \mathbb{B}$

(Un)Decidability

Existence

Halting Problem

$f : \mathbb{N}^m \rightarrow \mathbb{N}^n$

computable

uncomputable

computable

uncomputable
What can a computer do?

Run programs
What can a computer do?

Run programs
What can’t a computer do?

Can a computer:

- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
- Feel pain?
- Fall in love?

The question is can we phrase it as a function? and can we write a program?

If not, then a computer cannot solve the problem.
What can’t a computer do?

Can a computer:

- Understand emotions?
What can’t a computer do?

Can a computer:
- Understand emotions?
- Dream?
What can’t a computer do?

Can a computer:

- Understand emotions?
- Dream?
- Create beauty?
What can’t a computer do?

Can a computer:
- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
What can’t a computer do?

Can a computer:

- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
- Feel pain?
What can’t a computer do?

Can a computer:
- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
- Feel pain?
- Fall in love?
What can’t a computer do?

Can a computer:
- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
- Feel pain?
- Fall in love?

The question is can we phrase it as a function? and can we write a program?

If not, then a computer cannot solve the problem
A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is *computable* if, and only if, there is a while program $S$, such that for all states $s$, and $n \in \mathbb{N}$ with $s(x) = n$, then either there is a state $s'$ such that

$$< S, s > \Rightarrow^* s' \text{ and } s'(x) = f(n)$$

or $f(n)$ is undefined. Where $\Rightarrow^*$ is the *transitive closure* of $\Rightarrow$. 
Examples of Computable Functions

\( f(x) = 1 \) is computable using \( x := 1 \)

\( f(x) = x + 1 \) is computable using \( x := x + 1 \)

\[ f(x) = \begin{cases} 
-x & \text{if } x < 0 \\
 0 & \text{otherwise}
\end{cases} \]

is computable using if \( x < 0 \) then \( x := -x \)

Suppose \( f \) and \( g \) are computable functions. We can write a \texttt{while} program that computes \( f \circ g \) (Exercise).

In general we can construct computable functions from computable functions
Recall our pairing bijection $\varphi : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ from our previous work on coding. We will reuse this at various points in this chapter.

**Definition**

A function $f : \mathbb{N}^m \rightarrow \mathbb{N}^n$ – with $n, m \geq 1$ – is **computable** if, and only if, there is a function $g : \mathbb{N} \rightarrow \mathbb{N}$ which is computable in the sense of the previous Definition, such that

$$g(\varphi(x_1, \varphi(x_2, \ldots \varphi(x_{n-1}, x_n)))) = (\varphi(y_1, \varphi(y_2, \ldots \varphi(y_{m-1}, y_m) \ldots ))$$

where $\varphi$ is the pairing bijection, and,

$$f(x_1, x_2, \ldots, x_{n-1}, x_n) = (y_1, y_2, \ldots y_{m-1}, y_m)$$
Proving the Existence of Uncomputable Functions

The general idea:

1. There are a countable number of \texttt{while} programs
2. There are an uncountable number of functions

Therefore, there must be functions with no corresponding \texttt{while} program, so (by definition) those functions are not computable.

We can replace \texttt{while} by any language and just need to show that programs in that language of countable.

We now show (1) and (2) from above.
We can define a bijection $\phi_S : \text{Stm} \to \mathbb{N}$ to code statements in while into $\mathbb{N}$. All the details are in the notes and we just sketch the idea here.

Recall that the syntax of the while language is

\[
S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \\
b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \\
a ::= x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2
\]

To define $\phi_S$ we need a bijection that takes any $S$ and produces a natural number. First we need to deal with arithmetic and boolean expressions.
We code each of the cases

\[ \phi_A : \ AExp \rightarrow \mathbb{N} \]

\[ \phi_A(n) = 5 \times n \]
\[ \phi_A(x) = 1 + 5 \times x \]
\[ \phi_A(a_1 + a_2) = 2 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 - a_2) = 3 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 \times a_2) = 4 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]

What do we need to do for \( n \) and \( x \)?

Create a bijection with \( \mathbb{N} \)

Why does this work? (you need to understand for the exercises)

Example:

\[ \phi_A(1 + 1) = 2 + 5 \]
\[ \phi(5, 5) = 2 + 5(2 \times 5 + 1) - 1) = 1762 \]

1762 = 2 + 5 \( x \) (for \( x = 352 \)), all numbers of the form 2 + 5 \( x \) hold a
We code each of the cases

\[ \phi_A : \text{AExp} \rightarrow \mathbb{N} \]

\[ \phi_A(n) = 5 \times n \]
\[ \phi_A(x) = 1 + 5 \times x \]
\[ \phi_A(a_1 + a_2) = 2 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 - a_2) = 3 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 \times a_2) = 4 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]

What do we need to do for \( n \) and \( x \)? Create a bijection with \( \mathbb{N} \)
Coding Arithmetic Expressions

We code each of the cases

\[ \phi_A : \text{AExp} \rightarrow \mathbb{N} \]

\[ \phi_A(n) = 5 \times n \]
\[ \phi_A(x) = 1 + 5 \times x \]
\[ \phi_A(a_1 + a_2) = 2 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 - a_2) = 3 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 \times a_2) = 4 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]

What do we need to do for \(n\) and \(x\)? Create a bijection with \(\mathbb{N}\)

Why does this work? (you need to understand for the exercises)

Example: \(\phi_A(1 + 1) = 2 + 5\phi(5, 5) = 2 + 5(2^5(2 \times 5 + 1) - 1) = 1762\)

1762 = 2 + 5x (for x = 352), all numbers of the form 2 + 5x hold a \(a_1 + a_2\)
true and false only need one 'space' each. Note how we use $\phi$ and $\phi_A$.

$$\phi_B : \text{BExp} \rightarrow \mathbb{N}$$

$$\phi_B(\text{true}) = 0$$
$$\phi_B(\text{false}) = 1$$
$$\phi_B(a_1 = a_2) = 2 + 4 \times \phi(\phi_A(a_1), \phi_A(a_2))$$
$$\phi_B(a_1 \leq a_2) = 3 + 4 \times \phi(\phi_A(a_1), \phi_A(a_2))$$
$$\phi_B(\neg b) = 4 + 4 \times \phi_B(b)$$
$$\phi_B(b_1 \land b_2) = 5 + 4 \times \phi(\phi_B(b_1), \phi_B(b_2))$$
Extending the idea to Statements

Exercise

Using $\phi$, $\phi_A$ and $\phi_B$, complete the following recursive function $\phi_S$ which is a bijection from $\text{Stm}$ to $\mathbb{N}$:

$$
\phi_S : \text{Stm} \rightarrow \mathbb{N}
$$

$$
\begin{align*}
\phi_S(\text{skip}) &= 0 \\
\phi_S(\text{while } b \text{ do } S) &= 1 + 4 \times \phi(\phi_B(b), \phi_S(S)) \\
\phi_S(x := a) &= ? \\
\phi_S(S_1; S_2) &= ? \\
\phi_S(\text{if } b \text{ then } S_1 \text{ else } S_2) &= ?
\end{align*}
$$

Argue that this means there are only countably many programs in while.
Effectively countable and Gödel numbers

**Theorem**

The set of while programs is effectively countably infinite.

Effectively means we can write a program that could write them down in a list (enumerate them) and, conversely, find the index for each program.

**Definition**

To obtain the index, code number or Gödel number of a while program $S$, we will use $\gamma(S) = \phi_S(S)$ (the type of $\gamma$ is Stmt $\rightarrow$ N).

We now have

- a function to turn while programs into natural numbers, and
- a function to turn natural numbers into while programs i.e. $\gamma^{-1}$
There are uncountably many functions from $\mathbb{N} \to \mathbb{N}$.

We will use Cantor’s *Diagonalisation Method* to prove this.

Assume that there are countably many functions from $\mathbb{N} \to \mathbb{N}$ and therefore we can enumerate them

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>$f_0(0)$</td>
<td>$f_0(1)$</td>
<td>$f_0(2)$</td>
<td>$f_0(3)$</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_1(0)$</td>
<td>$f_1(1)$</td>
<td>$f_1(2)$</td>
<td>$f_1(3)$</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$f_2$</td>
<td>$f_2(0)$</td>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The function $f_{\text{new}}(n) = f_n(n) + 1$ must be different from every $f_k$

The enumeration is incomplete; we cannot count functions in $\mathbb{N} \to \mathbb{N}$
Uncountable Functions

There are uncountably many functions from $\mathbb{N} \to \mathbb{N}$.

We will use Cantor’s *Diagonalisation Method* to prove this.

Assume that there are countably many functions from $\mathbb{N} \to \mathbb{N}$ and therefore we can enumerate them.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>$f_0(0)$</td>
<td>$f_0(1)$</td>
<td>$f_0(2)$</td>
<td>$f_0(3)$</td>
<td>$f_0(4)$</td>
<td>...</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_1(0)$</td>
<td>$f_1(1)$</td>
<td>$f_1(2)$</td>
<td>$f_1(3)$</td>
<td>$f_1(4)$</td>
<td>...</td>
</tr>
<tr>
<td>$f_2$</td>
<td>$f_2(0)$</td>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>$f_2(4)$</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The function $f_{\text{new}}(n) = f_n(n) + 1$ must be different from every $f_k$.

The enumeration is incomplete; we cannot count functions in $\mathbb{N} \to \mathbb{N}$.
We have defined the notion of **computable functions**

We have proved the existence of **uncomputable functions**

We have done this in the context of **while** programs but the same arguments can be lifted to any programming language. As programs are necessarily finite, we can count them.

It turns out that the set of computable functions is the **same** for all realistic programming languages

Therefore, the notion of computable function is independent of the **while** language
A model of computation is universal if it can model itself. By this we mean we can write a program in the language that can simulate all other programs that can be written in the language.

To define a universal program we are going to use our previous trick of encoding programs as numbers. The idea will be to take that number as input then use $\gamma^1$ to lookup the relevant program and run it.
The function computed by the program $\gamma(i)$

**Definition**

Suppose that the program $S$ is the value of $\gamma^{-1}(i)$. Then we define $\eta_i$, the function associated with the $i$-th program as follows.

$$\eta_i(k) = \begin{cases} 
s'(x) & \text{if } \exists n, s'. \text{ with } < S, s[x \mapsto k] >\Rightarrow^n s' \\
\text{undefined} & \text{otherwise} \end{cases}$$

The undefined case allows for partial functions
\( \gamma(i) \) is a program whilst \( \eta_i \) is a function

**Lemma**

The function \( h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \) defined as \( h(i) = \eta_i \) is not injective.

Proof sketch:

- Suppose \( h(i) = h(j) \). If \( h \) were injective then \( i = j \)
- Let \( S = \gamma^{-1}(i) \) and let \( S' = \text{skip}; S \)
  \[
  \gamma(\text{skip}; S) = 2 + 4(2 \gamma(\text{skip})(2 \gamma(S) + 1) - 1)
  = 2 + 4((2i + 1) - 1)
  = 2 + 8i
  
  As \( 8i + 2 \neq i \) we do not have \( i = j \).

Each index \( i \) identifies a unique program but each function is computed by an infinite number of programs.
We define the Universal Function $\psi_U : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ for unary computable functions as:

$$\psi_U(a, b) = \eta_a(b)$$

where $a$ is the code number of function $\eta_a$. 
The Universal Program computes the Universal Function

The Universal Function $\psi_U(i, x)$ is computable

1. Find the **while** program associated with index $i$, which is $P_i = \gamma^{-1}(i)$. This is computable.

2. Next we simulate the execution of the program $P_i$, step-by-step. To do this we must code the reduction transitions using natural numbers. As **while** programs are numbers, these transitions are arithmetic operations, which are computable.

3. If and when the computation stops, the result will be in held in the value of variable $x$ in the final state.

The above steps describe the **universal program**

The definition and proof can be lifted to $n$-ary computable functions.
Church-Turing Thesis

Some things we should agree on:

- We can simulate all while programs using a single while program
- We can simulate all Java programs using a single Java program
- We can simulate all while programs using a Java program
- We can translate any Java program into an equivalent while program
- We can simulate all Java programs using a while program
- If we have a simulation in each direction then this means that the models of computation are equivalent

Thesis (Church-Turing)

Any sensible definition of computation will define the same functions to be computable as any other definition.
Reflective Questions

- Are there models of computation that are not equivalent to the while language? Hint: an automata can compute whether a number is divisible by 3, what else can it compute?

- What is the relationship between computability and tractability?

- If something is uncomputable does that mean we need to give up?

- What is the code number of the universal program, what happens if we pass it to itself? (Find out on Friday)