CO240 - Models of Computation

9th October 2019 — Lecture

Hilbert's Entscheidungproblem

Is there an algorithm, when fed any statement in the formal language of first-order logic, determines in a finite number of steps whether or not the statement is provable, using the usual rules of first-order logic?

From our first-order logic course, we know this isn't provable. What often happens in formal computer science, is that we think something holds, and end up not being able to prove it as the statement is false.

Entscheidungproblem means **decision problem**. Given a set S of finite data structures, such as formulae of first-order logic, and a property P of elements in S, such as whether the formulae is true or not, we have an associated decision procedure is to find an algorithm that terminates in 0, or 1, when given some $s \in S$, and gives the result $1 \Leftrightarrow P(s)$ (the property holds for the element).

Algorithms (Informal)

A question was to ask whether it was possible to prove if such an algorithm didn't exist. However, a formal definition of an algorithm is needed;

- finite description of the procedure as elementary operations
- deterministic; the next step is uniquely determined if there is one (note that we now have probabilistic programming, enough at the time as computers didn't even exist)
- may not terminate on some data, but we can get a result if it does

This was solved in the 1930s, by Alan Turing's Turing machines, and Church invented lambda calculus. Algorithms are regarded as data, and therefore can be passed on to another algorithm (we use this in compilers, etc.) which can process the algorithm passed as data, and reduced this to the Halting Problem.

Algorithms (formal)

Any formal definition of an algorithm must be;

- precise, meaning no assumptions, and preferably phrased in the language of mathematics
- simple, going for the absolute basicstyle
- general in the sense that it covers the whole span of algorithms

Turing discovered the **Universal Turing machine**, which takes in an input Turing machine, and some data. The universal machine acts as if it were the input Turing machine, operating on the given data, meaning that it can simulate an arbitrary Turing machine. The Church-Turing Thesis was a result of this, showing Turing machines were equivalent to Church's lambda calculus, thus anything computable can be computed by a Turing machine.

The Halting Problem

Given a set S of pairs (A, D), where A is an algorithm, and D is some input datum, $A(D)\downarrow$ holds for $(A, D) \in S$ if A applied to D eventually produces a result. This is unprovable, such that there is no algorithm H for all $(A, D) \in S$;

$$H(A, D) = \begin{cases} 1 & A(D)\downarrow \\ 0 & \text{otherwise} \end{cases}$$

We can go from the Halting Problem to Entscheidungproblem, in order to prove unsolvability. This is done by encoding pairs (A, D) of the Halting Problem as first-order logic statements $\Phi_{A,D}$ with the special property $\Phi_{A,D}$ is provable $\Leftrightarrow A(D)\downarrow$. Any algorithm that decides the provability of such statements is usable to decide the Halting Problem, therefore no such algorithm exists.

Hilbert's 10th Problem

A simpler proof of the Halting Problem uses Minsky and Lambek's register machines. The universal register machine, functions similar to how the Universal Turing machine, but with a register machine as input. This course is mainly on register machines, but it's important to know Turing machines for historical reasons.

Special Functions

A computable function is an algorithm that takes data, and sometimes gives a result (partial function). If it does terminate, then it gives this unique result. The question is whether it's possible to give a mathematical description of a computable function, as a special function between sets. At the end of the 1960s, Strachey and Scott in Oxford discovered it was possible to do so. **Denotational semantics** were introduced, describing the mathematical meaning of algorithms. Scott gave meaning to recursively defined algorithms as continuous functions between domains (sets with structure).

Semantics

```
power x n
  | n == 0    = 1
  | otherwise = x * power x (n - 1)
```

```
power' x n
  | n == 0 = 1
  | even n = k^2
  | odd n  = x*k^2
  where
    k = power' x (div n 2)
```

The first example, `power`, takes $O(n)$ steps to execute, whereas the second example, `power'`, takes $O(\log(n))$ steps. While the two functions are the same in terms of computable functions (since they give the same results), they are clearly different from an operational point of view. They are the same in terms of the high-level inputs and outputs, but aren't the same operationally. Operational semantics are the program's meaning in terms of the steps of computation taken.

Syntax of While

In the syntax below, we have $x \in \text{Var}$ to range over variables, and $n \in \mathbb{N}$ for the natural numbers. Note that the first item in C $(x := E)$ is an assignment to a variable.

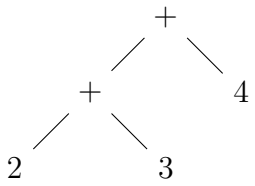
$$B \in \text{Bool} ::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid B \& B \mid \neg B \mid ...$$
$$E \in \text{Exp} ::= x \mid n \mid E + E \mid ...$$
$$C \in \text{Com} ::= x := E \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid C; C \mid \texttt{skip} \mid \texttt{while}\ B\ \texttt{do}\ C$$

Syntax of Simple Expressions

Similar to above, the $n \in \mathbb{N}$, and the operators are the same as mathematical operators. Here we will work with abstract syntax trees.

$$E \in \text{SimpleExp} ::= n \mid E + E \mid E \times E \mid ...$$

For example, we can draw out the AST for $(2+3)+4$ as below. Note that the $+$ and numbers in the tree are just syntax. While the brackets aren't needed in mathematics, they are required for the formal syntax tree.



The operational semantics for SimpleExp can be done in two ways; $E \Downarrow n$ (big-step / natural), which ignores the intermediate steps, and gives results immediately, or $E \to ... \to n$ (small-step / structural) semantics, which evaluates an expression step-by-step.

Big-step

Note that anything in violet is mathematical (hence $n \in \mathbb{N}$), and $+$ is actual numeric addition.

- (B-NUM) $$\frac{}{n \Downarrow n}$$
- (B-ADD) $$\frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}\ n_3 = n_1 + n_2$$

For example, we can prove $3 + (2 + 1) \Downarrow 6$, with the following derivation tree;

$$\dfrac{3 \Downarrow 3 \qquad \dfrac{2 \Downarrow 2 \qquad 1 \Downarrow 1}{2+1 \Downarrow 3}}{3 + (2 + 1) \Downarrow 6}$$

We have some properties on $\Downarrow$;

- **determinacy** $$\forall E, n_1, n_2 [E \Downarrow n_1 \wedge E \Downarrow n_2 \Rightarrow n_1 = n_2]$$

  this is the idea of something being deterministic, the same comment about probabilistic programming applies here too
- **totality** $$\forall E \exists n [E \Downarrow n]$$

  this holds for SimpleExp, but doesn't hold for the while language, as there can be a loop that doesn't terminate

Small-step

- (S-LEFT) $$\frac{E_1 \to E_1'}{E_1 + E_2 \to E_1' + E_2}$$
- (S-RIGHT) $$\frac{E \to E'}{n + E \to n + E'}$$
- (S-ADD) $n_3 = n_1 + n_2$ $$\frac{}{n_1 + n_2 \to n_3}$$

For example, consider the small-step evaluation of;

$$(2+3)+(4+1) \to 5+(4+1) \to 5+5 \to 10$$

Note that the **evaluation path**, as above, is not the same as the **derivation tree**.

Given a relation $\to$, we can define the reflexive transitive closure of $\to$ as $\to^*$. This has the rules such that $E \to^* E'$ holds directly (such that there are no steps of evaluation needed to get from E to E'), or that there is some finite sequence;

$$E \to E_1 \to E_2 \to ... \to E_k \to E'$$

We can say that n is the final answer of E if $E \to^* n$. While this definition is intuitive, the "..." in the sequence above is informal. Also, it is important to note that $E = E'$ is allowed when $E \to^* E'$, and therefore we can have $n \to^* n$, but $n \not\to n$, since the reflexive transitive closure can do 0, 1, or many steps. We say that some expression E is in **normal form**, and **irreducible** if $\neg\exists E'[E \to E']$. The normal form of expressions are numbers. Similar to $\Downarrow$, we also have some properties on $\to$;

- **determinacy** $\forall E, E_1, E_2[E \to E_1 \wedge E \to E_2 \Rightarrow E_1 = E_2]$

  with big-step, it was with respect to numbers, but here it is with respect to all the small computational steps
- **confluence** $\forall E, E_1, E_2[E \to^* E_1 \wedge E \to^* E_2 \Rightarrow \exists E'[E_1 \to^* E' \wedge E^2 \to^* E']]$
- **(strong) normalisation**

  there are no infinite sequences of expressions, which means that any evaluation path will eventually reach a normal form
- **theorem** $\forall E, n_1, n_2[E \to^* n_1 \wedge E \to^* n_2 \Rightarrow n_1 = n_2]$

The general theorem, coming back to the denotational semantics, is that $\forall E, n[E \Downarrow n \Leftrightarrow E \to^* n]$.

10th October 2019 — Tutorial

1. Find n such that $(4+1)+(2+2) \Downarrow n$

   When you do big-step evaluation, you go up, to the right, and then back down.

$$\dfrac{\dfrac{4 \Downarrow 4 \qquad 1 \Downarrow 1}{(4+1) \Downarrow 5} \qquad \dfrac{2 \Downarrow 2 \qquad 2 \Downarrow 2}{(2+2) \Downarrow 4}}{(4+1)+(2+2) \Downarrow 9}$$

2. Prove $(3+2) \times (1+4) \Downarrow 25$

$$\dfrac{\dfrac{3 \Downarrow 3 \qquad 2 \Downarrow 2}{(3+2) \Downarrow 5} \qquad \dfrac{1 \Downarrow 1 \qquad 4 \Downarrow 4}{(1+4) \Downarrow 5}}{(3+2) \times (1+4) \Downarrow 25}$$

3. Extending big-step semantics for subtraction;

$$\dfrac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_3 \Downarrow n_3}$$

   When $n_1 \geq n_2$, then $n_3 = n_1 - n_2$

$$\dfrac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_3 \Downarrow n_3}$$

   When $n_1 < n_2$, then $n_3 = ?$

   To handle the second case, we can deal with it in a number of ways, to keep it total. Below are a few methods, and their outcomes;

   - $= n_2 - n_1$

     this keeps it total, however can lead to unexpected behaviour as the number ends up positive; for example $(5-7)+10 = 12$
   - $= 0$

     this also keeps it total, however it also leads to different unexpected behaviour in the sense that $(5-7)+10 = 10$
   - $=$`NaN`

     if we have an error value, `NaN` in this case, we have to add rules to propagate this, which can be seen in the example below;

$$\dfrac{E_1 \Downarrow \texttt{NaN} \qquad E_2 \Downarrow n}{E_1 \pm E_2 \Downarrow \texttt{NaN}}$$

   - extend it to all numbers

4. Sound was broken for this part on Panopto.

   $((1+2)+(4+3)) \to 3+(4+3) \to 3+7 \to 10$

6. Suppose `SimpleExp` is extended with ?, such that $E \in \texttt{SimpleExp} ::= ... \mid (E \ \texttt{?} \ E)$,

   Note that the right hand side of $\Downarrow$ has to be a number, and therefore cannot be an expression. The example below captures the idea of non-determinism, as it can evaluate to either the left or the right expression.

   (a) Extending the big-step operation semantics to capture both $(1+2) \ \texttt{?} \ 4 \Downarrow 3$, and also the outcome $(1+2) \ \texttt{?} \ 4 \Downarrow 4$;

   $$\frac{E_1 \Downarrow n_1}{E_1 \ \texttt{?} \ E_2 \Downarrow n_1} \qquad \frac{E_2 \Downarrow n_2}{E_1 \ \texttt{?} \ E_2 \Downarrow n_2}$$

   (b) To give all possible derivation trees, in the case of this question, you just have to give all combinations of the left and right sides.

   (c) This isn't deterministic, as in the cases above, $E_1 \ \texttt{?} \ E_2$ can evaluate to either n_1 or n_2, and the two have no guarantee of being equal. However, it is total, as it will evaluate to something, given that all the expressions involved do evaluate to something (since it will always be one of two options).

16th October 2019 — Lecture

States

We define a state as a partial function, which is finite (despite our common assumption that resources are infinite). For example, we can look at a store like;

$$s = (a \mapsto 4, b \mapsto 3, ..., x \mapsto 4, y \mapsto 5, z \mapsto 6)$$

We can also denote an update to the store as such, where the value (x) is updated to 7, or is inserted into the store, if it didn't exist before.

$$S[x \mapsto 7](u) = \begin{cases} 7 & \text{if } u = x \\ s(u) & \text{otherwise} \end{cases}$$

Small-step semantics for the While language are defined with **configurations**, written in the form $\langle E, s\rangle$, $\langle B, s\rangle$, and $\langle C, s\rangle$, where the expressions E, B, and C are evaluated with respect to the state s. This, $\langle E, s\rangle$ can be read as "I want the behaviour of E in the context I have variable store s".

Small-step

It's important to note that these are very similar to the rules defined for SimpleExp, but with the addition of the (W-EXP.VAR) case, in which we look up values in the variable store.

- (W-EXP.LEFT) $$\frac{\langle E_1, s\rangle \to_e \langle E_1', s'\rangle}{\langle E_1 + E_2, s\rangle \to_e \langle E_1' + E_2, s'\rangle}$$
- (W-EXP.RIGHT) $$\frac{\langle E, s\rangle \to_e \langle E', s'\rangle}{\langle n + E, s\rangle \to_e \langle n + E', s'\rangle}$$
- (W-EXP.ADD) $n_3 = n_1 + n_2$ $$\frac{}{\langle n_1 + n_2, s\rangle \to_e \langle n_3, s\rangle}$$
- (W-EXP.VAR) $s(x) = n$ $$\frac{}{\langle x, s\rangle \to_e \langle n, s\rangle}$$

An important note from the lecturer is that the notation changes all the time, and can vary from text to text. It's much more important to gain an understanding of the concepts, rather than to memorise the notation itself. Consider the small-step evaluation for the following statement;

$$\langle (4+x)+(y+2), (x \mapsto 2, y \mapsto 3) \to_{\text{derivation}} \langle (4+2)+(y+3), s\rangle$$

The single $\rightarrow_{\text{derivation}}$ has to be done through the following steps, which we evaluate going up, to the right, and back down;

$$\frac{\dfrac{\langle x, s\rangle \rightarrow \langle 2, s\rangle}{\langle 4 + x, s\rangle \rightarrow \langle 4 + 2, s\rangle}}{\langle (4 + x) + (y + 2), s\rangle \rightarrow \langle (4 + 2) + (y + 2), s\rangle}$$

The full evaluation chain for the follow statement would be;

$$\langle (4+x)+(y+2), s\rangle \rightarrow \langle (4+2)+(y+2), s\rangle \rightarrow \langle 6+(y+2), s\rangle \rightarrow \langle 6+(3+2), s\rangle \rightarrow \langle 6+5, s\rangle \rightarrow \langle 11, s\rangle$$

The following rules can be extended to the booleans (the $E_1 = E_2$ case) as below; note that there are two cases to the numeric equality step. It is also important to note that when creating rules, the terminal states (`true` and `false`) in our case, are **never** on the left hand side of the step.

- LHS isn't fully evaluated ($E_1 = E_2$)

$$\frac{\langle E_1, s\rangle \rightarrow_e \langle E_1', s'\rangle}{\langle E_1 = E_2, s\rangle \rightarrow_b \langle E_1' = E_2, s'\rangle}$$

- LHS is fully evaluated ($n = E$)

$$\frac{\langle E, s\rangle \rightarrow_e \langle E', s'\rangle}{\langle n = E, s\rangle \rightarrow_b \langle n = E', s'\rangle}$$

- Both sides are fully evaluated ($n_1 = n_2$)

$$\frac{n_1 = n_2}{\langle n_1 = n_2, s\rangle \rightarrow_b \langle \texttt{true}, s\rangle} \qquad \frac{n_1 \neq n_2}{\langle n_1 = n_2, s\rangle \rightarrow_b \langle \texttt{false}, s\rangle}$$

We can also derive a similar set of rules for boolean composition ($B \& B$). Note that we are not doing any short-circuiting in our evaluation as that would require adding additional rules, however it is possible.

- LHS isn't fully evaluated ($B_1 \& B_2$)

$$\frac{\langle B_1, s\rangle \rightarrow_b \langle B_1', s'\rangle}{\langle B_1 \& B_2, s\rangle \rightarrow_b \langle B_1' \& B_2, s'\rangle}$$

- LHS is fully evaluated ($b \& B$)

$$\frac{\langle B, s\rangle \rightarrow_b \langle B', s'\rangle}{\langle b \& B, s\rangle \rightarrow_b \langle b \& B', s'\rangle}$$

- Both sides are fully evaluated ($b_1 \& b_2$), note that $b_3 = b_1 \wedge b_2$

$$\frac{}{\langle b_1 \& b_2, s\rangle \rightarrow_b \langle b_3, s\rangle}$$

For assignments, we have another set of rules, which update the variables in the store;

- (W-ASS.EXP)

$$\frac{\langle E, s\rangle \rightarrow_e \langle E', s'\rangle}{\langle x := E, s\rangle \rightarrow_c \langle x := E', s'\rangle}$$

- (W-ASS.NUM)

$$\frac{}{\langle x := n, s\rangle \rightarrow_c \langle \texttt{skip}, s[x \mapsto n]\rangle}$$

Similarly for sequential composition ($C; C$), we have the rules as follows;

- (W-SEQ.LEFT)

$$\frac{\langle C_1, s\rangle \rightarrow_c \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow_c \langle C_1'; C_2, s'\rangle}$$

- (W-SEQ.SKIP) $$\frac{}{\langle \texttt{skip}; C_2, s\rangle \to_c \langle C_2, s\rangle}$$

A more complex set of rules working on commands is the conditional case;

- (W-COND.TRUE) $$\frac{}{\langle \texttt{if true then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle C_1, s\rangle}$$
- (W-COND.FALSE) $$\frac{}{\langle \texttt{if false then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle C_2, s\rangle}$$
- (W-COND.BEXP) $$\frac{\langle B, s\rangle \to_b \langle B', s'\rangle}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \to_c \langle \texttt{if } B' \texttt{ then } C_1 \texttt{ else } C_2, s'\rangle}$$

The while case is trickier, and note that the following is **not** a computation step, but is rather unfolding the command;

- (W-WHILE) $$\frac{}{\langle \texttt{while } B \texttt{ do } C, s\rangle \to_c \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}, s}$$

16th October 2019 — Tutorial

1. Consider the evaluation of $(z := x; x := y); y := z$, and the start state $s = (x \mapsto 5, y \mapsto 7)$; We end up with the following derivation tree, for the first step;

$$\frac{\dfrac{\dfrac{\langle x, s\rangle \to_e \langle 5, s\rangle}{\langle z := x, s\rangle \to_c \langle z := 5, s\rangle}}{\langle z := x; x := y, s\rangle \to_c \langle z := 5; x := y, s\rangle}}{\langle (z := x; x := y); y := z, s\rangle \to_c \langle (z := 5; x := y); y := z, s\rangle}$$

2. There's a long question on (W-WHILE), but I'm too lazy to type it out.
3. In this question, we analyse operators which have side effects, such as the incrementing operator;

$$\text{(W-EXP.PP)}\quad \frac{}{\langle x\texttt{++}, s\rangle \to_e \langle n, s[x \mapsto n']\rangle}\ n = s(x), n' = n + 1$$

It first returns the value associated with x in the store s, and then reassigns x in the store to the original value $+1$.

For example, we can evaluate $x := (x\texttt{++}) + (x\texttt{++})$ as such;

$$\begin{aligned}\langle x := (x\texttt{++}) + (x\texttt{++}), (x \mapsto 2)\rangle &\to_c\\ \langle x := 2 + (x\texttt{++}), (x \mapsto 3)\rangle &\to_c\\ \langle x := 2 + 3, (x \mapsto 4)\rangle &\to_c\\ \langle x := 5, (x \mapsto 4)\rangle &\to_c\\ \langle \texttt{skip}, (x \mapsto 5)\rangle &\end{aligned}$$

17th October 2019 — Tutorial

5. In this question, we consider the ideas of parallelism, and interleaving;

- LHS evaluated first $$\frac{\langle C_1, s\rangle \to_c \langle C_1', s'\rangle}{\langle C_1 \mid\mid C_2, s\rangle \to_c \langle C_1' \mid\mid C_2, s'\rangle}$$
- RHS evaluated first $$\frac{\langle C_2, s\rangle \to_c \langle C_2', s'\rangle}{\langle C_1 \mid\mid C_2, s\rangle \to_c \langle C_1 \mid\mid C_2', s'\rangle}$$
- Collapsing after both sides are complete $$\frac{}{\langle \texttt{skip} \mid\mid \texttt{skip}, s\rangle \to_c \langle \texttt{skip}, s\rangle}$$

Note that this is an example of non-determinism, as we can see with the following example. Consider the program $(x := 1) \mid\mid (x := 2; x := x+2)$, with an initial state of $s = (x \mapsto 0)$, this can be evaluated to any of the three outcomes, where $x \mapsto 1$, 3, or 4.

6. Here we want to show that there can be C which doesn't hold for $\langle C, s\rangle \to_c^* \langle \texttt{skip}, s'\rangle$. Let $C = \texttt{while true do skip};$

$$\begin{aligned} \langle \texttt{while true do skip}, s\rangle &\to \\ \langle \texttt{if true } (\texttt{skip}; \texttt{while true do skip}), s\rangle &\to_c \\ \langle \texttt{skip}; \texttt{while true do skip}, s\rangle &\to_c \\ \langle \texttt{while true do skip}, s\rangle & \end{aligned}$$

Now let us assume that there is some finite evaluation path that takes n steps, such that $\langle C, s\rangle \to^* \langle \texttt{skip}, s'\rangle$, it follows that it can also take $n-3$ steps, as it takes 3 steps in our evaluation above. However, as this is deterministic, we have a contradiction.

23rd October 2019 — Lecture

For this lecture, we're using continuing to use the SimpleExp language for induction (refer back to **CO141**).

$$E \in \text{SimpleExp} ::= n \mid E + E \mid E \times E \mid ...$$

Our base case for this language would be $\forall n.P(n)$; (essentially we want to prove that P holds for an arbitrary $n \in \mathbb{N}$). Our first inductive case for this language would be to prove that $P(E_1 + E_2)$, assuming that $P(E_1)$, and $P(E_2)$ both hold. Similarly, for the second inductive case, we want to prove $P(E_1 \times E_2)$, with the same assumptions.

Determinacy

Formally, we want to prove the property on expressions $P(E) \triangleq \forall n_1, n_2.[E \Downarrow n_1 \wedge E \Downarrow n_2 \Rightarrow n_1 = n_2]$. This states that a simple expression cannot evaluate to more than one answer.

For the base case, we take an arbitrary n, and prove $P(n)$. We also take arbitrary n_1, n_2, and assume that $n \Downarrow n_1$, and also $n \Downarrow n_2$. However, with our rules for SimpleExp, the only way this can happen is if $n = n_1$, and also $n = n_2$, hence $n_1 = n_2$, therefore $P(n)$ holds for all n.

Now we can do the inductive case for $+$. We first assume that $P(E_1)$ and $P(E_2)$ hold, with the goal of proving $P(E_1 + E_2)$. We again take arbitrary n_1, n_2, and assume that $E_1 + E_2 \Downarrow n_1$ and $E_1 + E_2 \Downarrow n_1$. By pattern matching, with the case (B-ADD), we get the results that $E_1 \Downarrow n_{1,1}$ and $E_2 \Downarrow n_{2,1}$, with $n_{1,1} + n_{2,1} = n_1$, as well as $E_1 \Downarrow n_{1,2}$ and $E_2 \Downarrow n_{2,2}$, with $n_{1,2} + n_{2,2} = n_2$. However, due to our assumption that $P(E_1)$, and $P(E_2)$ both hold - we can say $n_{1,1} = n_{1,2}$, and $n_{2,1} = n_{2,2}$. Hence, we can conclude that $n_1 = n_2$.

The inductive case for $\times$ is similar.

23rd October 2019 — Tutorial

Binary Trees

$$bT ::= \texttt{Node} \mid \texttt{Branch}(bT, bT)$$

We also define the following functions on these trees;

$$\begin{aligned} \text{leaves}(\texttt{Node}) &= 1 \\ \text{leaves}(\texttt{Branch}(T_1, T_2)) &= \text{leaves}(T_1) + \text{leaves}(T_2) \\ \text{branches}(\texttt{Node}) &= 0 \\ \text{branches}(\texttt{Branch}(T_1, T_2)) &= \text{branches}(T_1) + \text{branches}(T_2) + 1 \end{aligned}$$

Our goal here is to prove $P(T) \triangleq \text{leaves}(T) = \text{branches}(T) + 1$ holds for all T.

The base case here is to prove $P(\texttt{Node})$, which is to prove leaves($\texttt{Node}$) = branches($\texttt{Node}$) + 1. By our function definitions, we have $1 = 0 + 1$, which is correct, hence we've proven the base case.

For the inductive step, we want to prove $P(\texttt{Branch}(T_1, T_2))$, which is to prove leaves($\texttt{Branch}(T_1, T_2)$) = branches($\texttt{Branch}(T_1, T_2)$) + 1. We are to assume $P(T_1)$ and $P(T_2)$ hold, which means that leaves(T_1) = branches(T_1) + 1, and similar for T_2. We use our function definitions to unfold leaves($\texttt{Branch}(T_1, T_2)$) = branches($\texttt{Branch}(T_1, T_2)$)+1, into leaves($T_1$)+leaves($T_2$) = branches($T_1$)+branches($T_2$)+1+1. However, with our assumptions that $P(T_1)$, and $P(T_2)$ hold, we can do substitutions to obtain branches(T_1) + 1 + branches(T_2) + 1 = branches(T_1) + branches(T_2) + 1 + 1 - which is valid. ■

Totality

This question refers back to SimpleExp.

For totality, we want to show $\forall E.P(E)$, where $P(E) \triangleq \exists n.E \Downarrow n$

Our base case is $E = n_1$, where n_1 is an arbitrary $\mathbb{N}$. To prove $P(n_1)$, we have $\exists n.n_1 \Downarrow n$. By the rule (B-NUM), we have $\dfrac{}{n_1 \Downarrow n_1}$, and therefore our n is n_1.

Our inductive case + (ignoring $\times$, as it is similar), is to prove $P(E_1 + E_2)$, which is $\exists n.E_1 + E_2 \Downarrow n$. By our inductive hypothesis, we are have $P(E_1) \wedge P(E_2)$, hence $\exists n_1.E_1 \Downarrow n_1 \wedge \exists n_2.E_2 \Downarrow n_2$. By using (B-ADD), we have $n_3 = n_1 + n_2$;

$$\frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}$$

24th October 2019 — Lecture

Note that the slides weren't recorded on Panopto.

Determinacy

We first define the property as follows;

$$P(E) \triangleq \forall E_1, E_2.E \to E_1 \wedge E \to E_2 \Rightarrow E_1 = E_2$$

Our base case is to take $E = n$, where n is an arbitrary $\mathbb{N}$. However, note that we don't actually need to do anything to prove $P(n)$, as there aren't any steps to do from n, therefore the LHS of the implication never happens.

In the inductive + case, we take $E = E_1' + E_2'$. Our inductive hypothesis allows us to assume $P(E_1') \wedge P(E_2')$. We first assume that $E_1' + E_2' \to E_1 \wedge E_1' + E_2' \to E_2$, for arbitrary E_1, E_2. From here, we can analyse each case, and pattern match those to the small-step evaluation rules;

- $E_1' = n_1$ and $E_2' = n_2$ (S-ADD)

  Hence $E_1 = n_1 + n_2 = n_3 = E_2$
- $E_1' = n_1$ and $E_2' \neq n_2$ (S-RIGHT)

  $E_2' \to E_2''$, and $E_1 = n_1 + E_2''$

  $E_2' \to E_2'''$, and $E_2 = n_1 + E_2'''$

  By our inductive hypothesis $P(E_2')$, we have $E_2'' = E_2''$ hence $E_1 = E_2$
- $E_1' \neq n$ (S-LEFT)

  $E_1' \to E_1''$, and $E_1 = E_1'' + E_2'$

  $E_1' \to E_1'''$, and $E_2 = E_1''' + E_2'$

  By our inductive hypothesis $P(E_1')$, we have $E_1'' = E_1''$ hence $E_1 = E_2$

Confluence

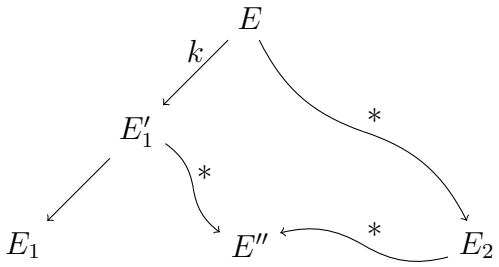
We first define the property as follows;

$$P(n) \triangleq \forall E_1, E_2, E.E \to_n E_1 \wedge E \to^* E_2 \Rightarrow \exists E'.E_1 \to^* E' \wedge E_2 \to^* E'$$

Note that we aren't doing structural induction, but rather mathematical induction on n.

In our base case we want to prove $P(0)$. As $n = 0$, we have $E = E_1$, and therefore we can let $E' = E_2$. This is justified such that we can do no steps to get from E to E_1, and then however many steps from there to E_2 - better shown in the diagram below;

In the inductive case, we want to prove $P(k+1)$, assuming $P(k)$. Because we're in a case where there's one or more steps, we can add a E'_1 in our path, which is k steps away from E. As we've assumed $P(k)$, it follows that there's some E'', such that $E'_1 \to^* E'' \wedge E_2 \to^* E''$. This can be shown in the following diagram;



Now we can analyse the two cases;

- $E'_1 = E''$

  As we have $E_2 \to^* E''$, we get $E_2 \to^* E_1$, hence we can say $E' = E_1$.

- $E'_1 \neq E''$

  As they aren't the same, we can add a step in the path, hence $E'_1 \to E_1 \to^* E''$ (E_1 must be the next step due to determinacy). Since we get $E_1 \to^* E''$, we can say $E' = E''$.

30th October 2019 — Lecture

SimpleExp

In this part of the lecture, our goal is to connect $\Downarrow$, and $\to^*$ for SimpleExp. Formally, we want to show;

$$\forall E, n.E \Downarrow n \Leftrightarrow E \to^* n$$

We first do the direction from $\Downarrow$ to $\to^*$, with the property $P(E) \triangleq \forall n.E \Downarrow n \Rightarrow E \to^* n$;

There is almost no work to be done in the base case, where $E = n$, for some arbitrary $n \in \mathbb{N}$, as the big-step rule states that $n \Downarrow n$, and it takes 0 steps for n to get to itself (in small-step).

For the inductive + case, once again we are ignoring $\times$ as the results are quite similar, we take $E = E_1 + E_2$, with the inductive hypothesis $P(E_1) \wedge P(E_2)$. If we take some arbitrary n, such that $E \Downarrow n$, our rules for big-step require there to be some n_1, n_2 which satisfy $E_1 \Downarrow n_1$, $E_2 \Downarrow n_2$, and $n = n_1 + n_2$ (B-ADD). Additionally, due to our inductive hypothesis, we also get $E_1 \to^* n_1$, and also $E_2 \to^* n_2$. Since we have $E_1 \to^* n_1$, we can apply lemma 1, using (S-LEFT), r times, such that we have $E_1 + E_2 \to^* n_1 + E_2$. Similarly with lemma 2, using (S-RIGHT), we have $E_1 + E_2 \to^* n_1 + n_2$. Finally, due to (S-ADD), $E_1 + E_2 \to^* n$.

While

In the second part of the lecture, we're revisiting the rules for the While language, which has the BNFs as follows;

$$B \in \text{Bool} ::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid B \& B \mid \neg B \mid ...$$
$$E \in \text{Exp} ::= x \mid n \mid E + E \mid ...$$
$$C \in \text{Com} ::= x := E \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \mid C; C \mid \texttt{skip} \mid \texttt{while } B \texttt{ do } C$$

This has the following big-step rules;

- $$\frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle x := E, s\rangle \Downarrow \langle s'[x \mapsto n]\rangle}$$
- $$\frac{\langle C_1, s\rangle \Downarrow \langle s'\rangle \qquad \langle C_2, s'\rangle \Downarrow \langle s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle s''\rangle}$$
- $$\frac{\langle B, s\rangle \Downarrow \langle \texttt{true}, s'\rangle \qquad \langle C_1, s'\rangle \Downarrow \langle s''\rangle}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \Downarrow \langle s''\rangle}$$
- $$\frac{\langle B, s\rangle \Downarrow \langle \texttt{false}, s'\rangle \qquad \langle C_2, s'\rangle \Downarrow \langle s''\rangle}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \Downarrow \langle s''\rangle}$$
- $$\frac{\langle B, s\rangle \Downarrow \langle \texttt{true}, s'\rangle \qquad \langle C, s'\rangle \Downarrow \langle s''\rangle \qquad \langle \texttt{while } B \texttt{ do } C, s''\rangle \Downarrow \langle s'''\rangle}{\langle \texttt{while } B \texttt{ do } C, s\rangle \Downarrow \langle s'''\rangle}$$
- $$\frac{\langle B, s\rangle \Downarrow \langle \texttt{false}, s'\rangle}{\langle \texttt{while } B \texttt{ do } C, s\rangle \Downarrow \langle s'\rangle}$$

After this point, there isn't any sound on the Panopto recording; the remainder of the lecture goes over heaps found in the coursework.

6th November 2019 — Tutorial

The lecture starts off going over questions in the coursework. Note that the question concerning $\neg$ can be done in a single case, with a side condition;

$$\frac{\langle B, s, h\rangle \Downarrow \langle b, s', h'\rangle}{\langle \neg B, s, h\rangle \Downarrow \langle b', s', h'\rangle}\ b' = \neg b$$

The next part concerns question 1 from last year's exam, which involves the language ForWithBreak;

$$C ::= \texttt{skip} \mid \texttt{break} \mid x := E \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \mid \texttt{for } x \texttt{ in } [E, E] \texttt{ do } C \mid C; C$$
$$\mathcal{O} ::= \mathcal{N} \mid \mathcal{B}$$

Writing out the big-step rules is too much effort; refer to the exam paper for the trees.

The `for` loop iterates between the values in the range, substituting x for each of the values iteratively, to be used in C. The way `break` works is by terminating sequential commands; for example, if we had $C = \texttt{break}; C_2$, it would evaluate to $\mathcal{B}$, and skipping the execution of C_2. However, if a `break` is encountered within a `for` loop, such that $\langle C, s\rangle \Downarrow_c \langle \mathcal{B}, s'\rangle$, we terminate the loop, and then the program continues normally, such that the loop command evaluates to $\langle \mathcal{N}, s'\rangle$.

7th November 2019 — Lecture

Register Machines

In a register machine, we operate on registers with idealised natural numbers (hence we don't consider physical limitations such as overflow). The elementary operations available to us are as follows;

- add 1 to a register
- jumps
- test whether a register is 0 (+ conditionals)
- subtract 1 from a non-zero register

Formally, a register machine is specified with finitely many registers $(R_0, R_1, \ldots, R_n)$, and a finite list of instructions, of the following form;

label: body	description
$L_i : R^+ \to L'$	Add 1 to the contents of R, and jump to L'
$L_i : R^- \to L', L''$	If $R > 0$, subtract 1 from R, and jump to L', else jump to L''
`HALT`	Terminate the program

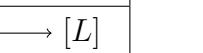
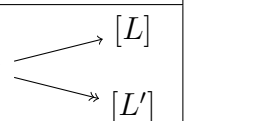
A register machine's configuration is written in the following form;

$$c = (\ell, r_0, \ldots, r_n)$$

We define a computation of a register machine as a sequence of configurations; $c_0, c_1, \ldots$, where c_0 is the initial configuration, and c_{i+1} is determined by the previous (c_i) configuration. This sequence **can** be infinite, if the program does not halt. A program halts if it reaches the `HALT` instruction (a **proper** halt), or there is a jump to an instruction that does not exist (an **erroneous** halt).

Graphical Representation

We can also represent these instructions graphically, instead of as a list of instructions. Note that $[L]$ means the body of the instruction with label L.

instruction	representation
$R^+ \to L$	
$R^- \to L, L'$	

Partial Functions

We define a partial function from a set X to a set Y, as any subset $f \subseteq X \times Y$, such that $(x, y) \in f \wedge (x, y') \in f \Rightarrow y = y'$.
Additional notation is as follows;

- $f(x) = y$ $\qquad (x, y) \in f$
- $f(x) \downarrow$ $\qquad \exists y \in Y.[f(x) = y]$

  x maps to something in y, under f
- $f(x) \uparrow$ $\qquad \neg\exists y \in Y.[f(x) = y]$

  x doesn't map to something in y, under f
- $X \rightharpoonup Y$ is the set of all **partial** functions from X to Y, whereas $X \to Y$ is the set of all **total** functions

  a partial function f is total if $\forall x \in X.[f(x) \downarrow]$

We define a function $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ as computable if there is a register machine with at least $n + 1$ registers such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$, starting at the configuration $c = (0, 0, x_1, \ldots, x_n, 0, \ldots, 0)$ halts with $R_0 = y$ if and only if $f(x_1, \ldots, x_n) = y$.

Register machines can be constructed to prove the computability of functions. For example, multiplication can be done with an auxiliary register; every time x is reduced by 1, y is copied into the auxiliary register, as well as R_0, and then y is restored from the auxiliary register.

13th November 2019 — Lecture

Numerical Coding of Pairs

The goal is to get a bijection between pairs of natural numbers, and the natural numbers themselves, for example the representation gives us the equivalence $27 = \texttt{0b11011} = \langle\!\langle 0, 13 \rangle\!\rangle = \langle 2, 3 \rangle$;

$$\langle\!\langle x, y \rangle\!\rangle \triangleq 2^x(2y+1) \qquad \text{bijection between } \mathbb{N} \times \mathbb{N} \text{ and } \mathbb{N}^+$$
$$\langle x, y \rangle \triangleq 2^x(2y+1) - 1 \qquad \text{bijection between } \mathbb{N} \times \mathbb{N} \text{ and } \mathbb{N}$$

We can convert $\langle\!\langle x, y \rangle\!\rangle$ to binary as first y in binary, followed by a $\texttt{1}$, and then followed by x $\texttt{0}$s, as we shift it by x bits. Similarly, $\langle x, y \rangle$ can be thought of as y in binary, followed by a $\texttt{0}$, and then x $\texttt{1}$s (since we can subtract $\texttt{1}$ from $\texttt{1}$ and x $\texttt{0}$s).

Numerical Coding of Lists

List $\mathbb{N}$ is defined as the set of all finite lists of natural numbers, with the following rules;

- the empty list: $[] \in \mathit{List}\ \mathbb{N}$
- the cons list: $x \in \mathbb{N} \wedge \ell \in \mathit{List}\ \mathbb{N} \Rightarrow x :: \ell \in \mathit{List}\ \mathbb{N}$
- shorthand notation: $[x_1, x_2, \ldots, x_n] \triangleq x_1 :: (x_2 :: (\cdots :: (x_n :: [])\ldots)$

Our notation will use $\ulcorner \ell \urcorner \in \mathbb{N}$ to represent the natural number representing the list $\ell \in \mathit{List}\ \mathbb{N}$;

$$\ulcorner [] \urcorner \triangleq 0$$
$$\ulcorner x :: \ell \urcorner \triangleq \langle\!\langle x, \ulcorner \ell \urcorner \rangle\!\rangle = 2^x(2 \cdot \ulcorner \ell \urcorner + 1)$$

Therefore, we can say $\ulcorner [x_1, x_2, \ldots, x_n] \urcorner = \langle\!\langle x_1, \langle\!\langle x_2, \langle\!\langle \ldots \langle\!\langle x_n, 0 \rangle\!\rangle \ldots \rangle\!\rangle \rangle\!\rangle$

We can also obtain the following result;

$$\texttt{0b}\ulcorner [x_1, x_2, \ldots, x_n] \urcorner = \texttt{1}\underbrace{\texttt{0}\cdots\texttt{0}}_{x_n\ \texttt{0s}}\texttt{1}\underbrace{\texttt{0}\cdots\texttt{0}}_{x_{n-1}\ \texttt{0s}}\cdots\texttt{1}\underbrace{\texttt{0}\cdots\texttt{0}}_{x_1\ \texttt{0s}}$$

Numerical Coding of Programs

Consider some register machine program P;

$$L_0 : b_0$$
$$L_1 : b_1$$
$$\vdots$$
$$L_n : b_n$$

The program's encoding is $\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner b_0 \urcorner, \ulcorner b_1 \urcorner, \ldots, \ulcorner n_n \urcorner] \urcorner$. The bodies can be encoded as such;

$$\ulcorner R_i^+ \to L_j \urcorner \triangleq \langle\!\langle 2i, j \rangle\!\rangle$$
$$\ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \langle\!\langle 2i+1, \langle j, k \rangle \rangle\!\rangle$$
$$\ulcorner \texttt{HALT} \urcorner \triangleq 0$$

14th November 2019 — Lecture

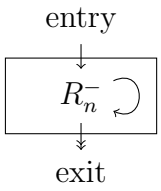
Gadgets

We can think of these building blocks as subroutines. These gadgets have one entry point, but can have multiple exit points. These gadgets can use scratch registers, assuming they are initialised to 0,

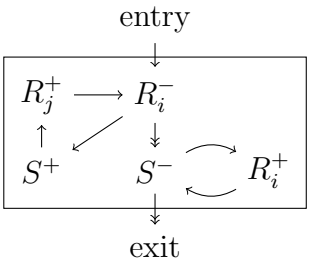
but must reset them back to 0 after execution. Gadgets can also use other registers (however they must be specified), and we can use them to contain arguments to the gadgets, the start initialisation of the gadget, or the result of what the gadget is producing.

Here are some examples of gadgets, and their graphical representations;

- zero R_n $R_n = 0$

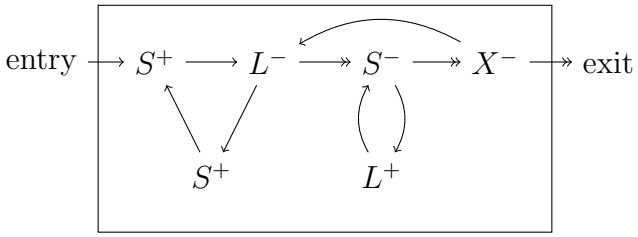


- add R_i to R_j $R_j = R_j + R_i$



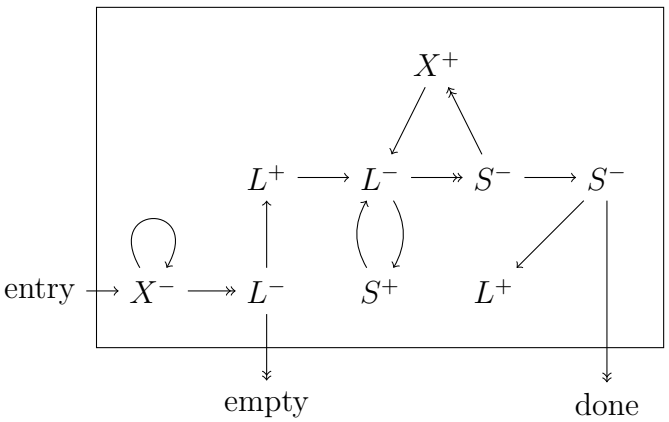
With these gadgets, we can then chain them to produce other gadgets. For example, we can copy the value of R_i into R_j (such that R_j's old value is replaced with R_i's), by first zeroing R_j with the first gadget, and then adding R_i to it, with the second gadget.

Another important gadget is one to perform the push operation, which essentially performs $x :: \ell$. The following gadget pushes X to L, such that $X = x, L = \ell, X' = 0, L' = \langle\langle x, \ell \rangle\rangle = 2^x(2\ell + 1)$, where X' and L' are the values of X and L after the gadget, respectively. These work with the numerical coding of lists. The proof for this is in the notes, in *Lecture 6 - Universal RM.pdf*.



20th November 2019 **Lecture**

Continuing on from last week's lecture, we look at the gadget to perform the pop operation, which has two outcomes. In the first outcome, where the list is empty $L = \ell = 0$, then we return with $X' = 0$. On the other hand, we return $X' = x$, and $L' = \ell$, if the initial state is $L = \langle\langle x, \ell \rangle\rangle$.



The Universal Register Machine

This gadgets used in the universal register machine are **copy**, **push**, and **pop**. The universal register machine is an interpreter of the Godel number of programs. It starts with $R_0 = 0$, which is designed to be the return value of the program, $R_1 = e = \ulcorner P \urcorner$ (the code of a program), and $R_2 = a = \ulcorner [a_1, \ldots, a_n] \urcorner$ (a list of arguments). The goal of the universal register machine is to carry out the computation of P starting with $R_0 = 0, R_1 = a_1, \ldots R_n = a_n$, and all other registers at 0. The mnemonics of the registers in U are as follows (anything else can be used as a scratch register);

$R_0 \equiv$???	result of simulated computation (if any)
$R_1 \equiv P$	**p**rogram code of the register machine to be simulated
$R_2 \equiv A$	list of **a**rguments / register contents of the simulated machine
$R_3 \equiv PC$	**p**rogram **counter** - current instruction label
$R_4 \equiv N$	label number(s) of the **n**ext instruction(s), also code for current instruction
$R_5 \equiv C$	code of the **c**urrent instruction body
$R_6 \equiv R$	value of the **r**egister to be used by current instruction
$R_7 \equiv S$	auxiliary register
$R_8 \equiv T$	auxiliary register

The overall execution structure of the universal register machine is as follows;

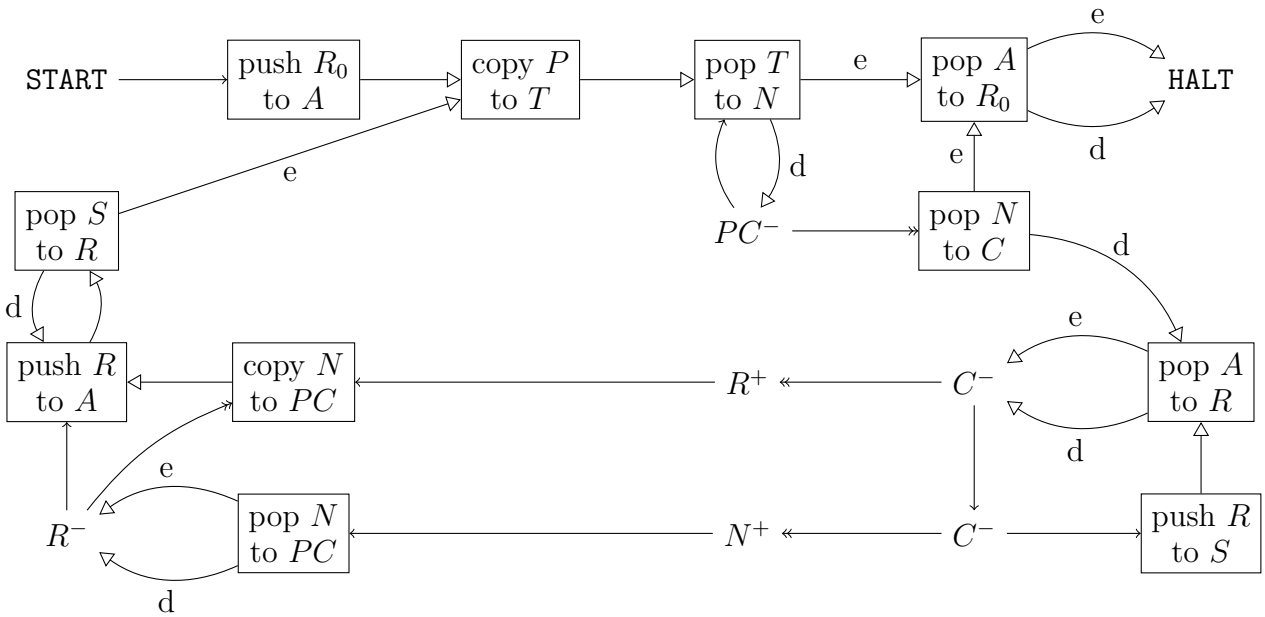
1. copy PC^{th} instruction of P to N (halt if $PC >$ length of list), and go to step 2.
2. if $N = 0$, then halt, otherwise $N = \langle\langle y, z \rangle\rangle$, then set $C = y, N = z$, and go to step 3.

   if $C = 2i$ then instruction is $R_i^+ \to L_z$

   if $C = 2i + 1$ then instruction is $R_i^- \to L_j, L_k$, where $z = \langle j, k \rangle$

   note that in this case, when we say R_i, we're referring to the register in the **simulated** machine not the **universal** machine.
3. copy i^{th} item of A to R, and go to step 4
4. execute the current instruction on R, update the PC to the next label, restore the register value to A, and go to step 1

The universal register machine can be represented graphically as below. For a full explanation of how the steps work in detail, the lecture explains it well. Note that the single N^+ converts between the single and double angle encodings, since the single angle encoding is used for $\langle j, k \rangle$.



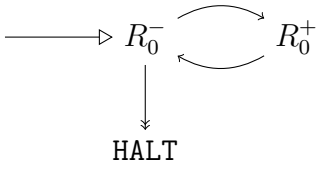
Halting Problem

We define a register machine H to decide the Halting Problem, if for all $e, a_1, \ldots, a_n \in \mathbb{N}$, starting H with the configuration $(\ell, 0, e, \ulcorner[a_1, \ldots, a_n]\urcorner)$ (where ℓ is some starting label), the computation of H always halts with $R_0 = 0$ or 1. $R_0 = 1$ iff the register machine with index e eventually halts with $R_0 = 0, R_1 = a_1, \ldots, R_n = a_n$. Clearly, simulation doesn't work; if the program with index e doesn't halt, then H cannot halt if it simulates it.

We can prove this machine cannot exist by contradiction. Let us first add two steps, after the program starts - we name this new machine H';

1. $Z = R_1$ — Z is some unused register
2. push Z to R_2 — adding the program as an argument

Then, every normal `HALT` or erroneous halt is replaced with the following loop;



Let us now call this new register machine, based off of H', C, with program index $c \in \mathbb{N}$.

As such, C starting with $R_1 = c$ eventually halts iff H' starting with $R_1 = c$ halts with $R_0 = 0$ iff H starting with $R_1 = c, R_2 = \ulcorner[c]\urcorner$ halts with R_0 iff C starting with $R_1 = c$ does not halt.

21st November 2019 — Lecture

For each $e \in \mathbb{N}$, let $\phi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function computed by the register machine with index e. $\phi_e(x) = y$ holds iff the register machine halts with $R_0 = y$, after starting with $R_0 = 0, R_1 = x$.

Let $f \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function $\{(x, 0) \mid \phi_x(x) \uparrow\}$, such that

$$f(x) = \begin{cases} 0 & \text{if } \phi_x(x) \uparrow \text{ (uncomputable)} \\ \text{undefined} & \text{if } \phi_x(x) \downarrow \text{ (computable)} \end{cases}$$

If we assume that f is computable, then $f = \phi_e$, for some $e \in \mathbb{N}$. We can consider each case;

- $\phi_e(e) \uparrow$, then $f(e) = 0$, by definition of f, which therefore means $f\phi_e(e) = 0$, but then it is computable hence $\phi_e(e) \downarrow$
- $\phi_e(e) \downarrow$, then $f(e) =$ undefined, hence $\phi_e(e) \uparrow$

As both cases lead to a contradiction, the entire assumption fails, hence f isn't computable.

A similar application can be done to sets; take a subset $S \subseteq \mathbb{N}$, the **characteristic function** of S, $\chi_S \in \mathbb{N} \to \mathbb{N}$, is defined as follows;

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

A subset S is called RM decidable if its characteristic function χ_S is a register machine computable function. Therefore, S is decidable if there is some register machine M such that for all $x \in \mathbb{N}$, M eventually terminates with $R_0 = \chi_S(x)$, when starting with $R_0 = 0, R_1 = 1$.

Turing Machines

Informally a Turing machine is a head with a state, on a tape with symbols. It has the following characteristics;

- the machine starts in state s with the tape head pointing to the first symbol of the finite input string (everything else is blank)
- the machine computes in steps, depending on the current state of the head, and the symbol being scanned by the tape head
- an action is to overwrite the current tape cell with a symbol, move left or right one cell, and change state
- finitely many states
- finitely many symbols

Formally, a Turing machine is specified by a quadruple $M = (Q, \Sigma, s, \delta)$, where;

- Q a finite set of machine states
- Σ a finite set of tape symbols (containing a **blank** - $\sqcup$)
- $s \in Q$ an initial state
- $\delta \in (Q \times \Sigma) \rightharpoonup (Q \times \Sigma \times \{L, R\})$ a partial transition function

  this takes the current state, the current symbol, and possibly creates the next state, the next symbol, and a movement to the left or right

  note that it is a partial function; if it reaches something it cannot do, the machine halts

We can then formally define a configuration (q, w, u) as;

- $q \in Q$ the current state
- $w \in \Sigma^*$ a finite (possibly empty) string of tape symbols to the left of the head
- $u \in \Sigma^*$ a finite (possibly empty) string of tape symbols under and to the right of the head

An initial configuration is (s, ϵ, u), for an initial state, and a string of tape symbols u.

We can define elementary functions $\text{first} : \Sigma^* \to \Sigma \times \Sigma^*$, and $\text{last} : \Sigma^* \to \Sigma \times \Sigma^*$ as follows, which split off the first and last symbols of a string.

$$\text{first}(w) = \begin{cases} (a, v) & \text{if } w = av \\ (\sqcup, \epsilon) & \text{if } w = \epsilon \end{cases}$$

$$\text{last}(w) = \begin{cases} (a, v) & \text{if } w = va \\ (\sqcup, \epsilon) & \text{if } w = \epsilon \end{cases}$$

From this, we can define the computation of a Turing Machine $M = (Q, \Sigma, s, \delta)$, $(q, w, u) \to_M (q', w', u')$ by;

- $$\frac{\text{first}(u) = (a, u') \qquad \delta(q, a) = (q', a', L) \qquad \text{last}(w) = (b, w')}{(q, w, u) \to_M (q', w', ba'u')}$$
- $$\frac{\text{first}(u) = (a, u') \qquad \delta(q, a) = (q', a', R)}{(q, w, u) \to_M (q', wa', u')}$$

If (q, w, u) has no computation step, it is a **normal form**, this happens when $\delta(q, a)$ isn't defined for $\text{first}(u) = (a, u')$. The computation for a Turing Machine M is a sequence of configurations $c_0, c_1, c_2, \ldots$, where $c_0 = (s, \epsilon, u)$ is an initial configuration, and $c_i \to_M c_{i+1}$ holds for all i. This computation does not halt if the sequence is infinite, but does halt if the sequence is finite and the last element is a normal form.

27th November 2019 — Lecture

The λ-Calculus

This can be considered the "simplest programming language", where it is defined as follows;

$$M ::= x \mid \lambda x.\ M \mid M\ M$$

- x — a variable
- $\lambda x.M$ — an abstraction (single-parameter function, where M is the body)
- $M\ M$ — an application

For example, $\lambda x.\ (2x + 1)$ is equivalent to `x => 2*x + 1` in Scala.

Syntax

A variable is bound in M if it has a corresponding λ (binder), for example (bound variables in red, free variables in blue);

- $\lambda x.\ x$
- $\lambda x.\ y$
- $\lambda x.\ \lambda y.\ \lambda z.\ xy$ — with **contraction**, it can also be written as $\lambda xyz.\ xy$
- $((\lambda x.\ xy)(\lambda y.\ xy))(\lambda xy.\ xyz)$ — application is **left-associative**
- $(\lambda x.\ (\lambda y.\ xy)y)(\lambda z.\ zx)$

A **closed term** has no free variables. Formally, we can define free variables on the structure of M as follows;

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} && \text{all variables are free by themselves} \\ \mathrm{FV}(\lambda x.\ M) &= \mathrm{FV}(M) \setminus \{x\} \\ \mathrm{FV}(MN) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{aligned}$$

In terms of programming languages, we can consider bound variables as function parameters.

```
function (x, y) {
    return x + y;
}
```

Can be written as $\lambda xy.\ x + y$ (assuming we had some sort of arithmetic plus operator). On the other hand, free variables are "declared" in a higher-up scope, for example;

```
function (x, y) {
    return (x + y) % z;
}
```

Tutorial

(a) Doing this as one question; the colours are as follows; binding occurrences, bound occurrences, and free occurrences;

$(\lambda xy.\ y(\lambda x.\ xy)z)(x(\lambda zx.\ xzy))$

(b) Not really bothered to write this out, the main point is to consider scoping.

α-equivalence

The process of renaming bound variables is called α-equivalence. Syntactically, the functions on lines 1 and 2 are different, but they are semantically the same;

```
1 function(x, y) { return x + y; }
2 function(a, b) { return a + b; }
```

The functions can be represented as the following λ-terms, omitting the plus; $\lambda xy.\ xy$ and $\lambda ab.\ ab$ respectively. In the λ-calculus, they are the same, despite being represented differently, they are computed the same, therefore;

$$\lambda xy.\ xy =_\alpha \lambda ab.\ ab$$

Intuitively, $M =_\alpha N$ iff one can be obtained from the other (obviously both ways) by reaming the bound variables. They must also have the same set of free variables. The general strategy is as follows;

- Are they of the same structure?
- Do all of the free variables match?
- Can you rename the bound variables so that they match?

Substitution

In the λ-calculus, we use the syntax $M[N/x]$, to replace x with N in M.

- $x[y/x]$ y

  the x is replaced with a y
- $z[y/x]$ z

  there is no x to replace
- $(xy)(yz)[y/x]$ $(yy)(yz)$

  substitution occurs in both sub-parts
- $(\lambda z.\ xz)[y/x]$ $\lambda z.\ yz$

  x is free
- $(\lambda x.\ xy)[y/x]$ $\lambda x.\ xy$

  in α-equivalence, the bound x can be renamed to anything, and therefore no substitution can be done, since this is the same as $(\lambda t.\ ty)[y/x]$
- $(\lambda y.\ xy)[y/x]$ $\lambda z.\ yz$

  rename the bound variable (and binder)

Formally, this is set out as;

$$x[M/y] = \begin{cases} M & x = y \\ x & x \neq y \end{cases}$$

$$(\lambda x.\ N)[M/y] = \begin{cases} \lambda x.\ N & x = y \\ \lambda z.\ N[z/x][M/y] & x \neq y \end{cases} \qquad z \notin \mathrm{FV}(N) \setminus \{x\}, z \notin \mathrm{FV}(M), z \neq y$$

This new z is not in the free variables of N, is not in the free variables of M (otherwise it would bind). It cannot be y, but it can be x, as we don't have to rename all the time. In short it must be different from all the free variables you can see in this expression.

$$(M_1 M_2)[M/y] = (M_1[M/y])(M_2[M/y])$$

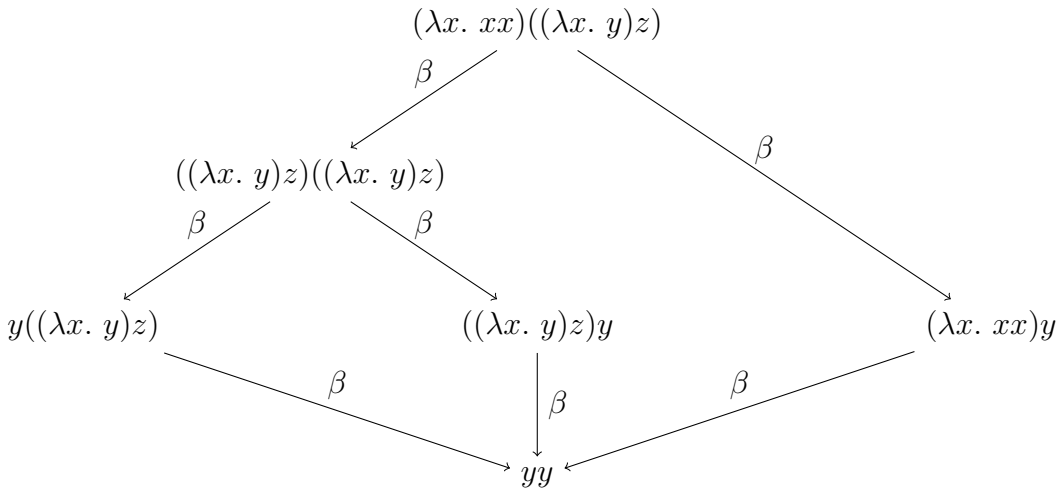
β-reduction

For computation, we have the following rules. Note that we don't need to go into this much detail when we do our reductions, as they tend to be quite clear. The final rule allows for us to use α-equivalences in our reduction.

- $\dfrac{}{(\lambda x.\ M)N \to_\beta M[N/x]}$ take this N to be the function parameter, put it in the body

  An abstraction followed by an application is called a redex - it can be reduced (the part before the $\to_\beta$).
- $\dfrac{M \to_\beta M'}{\lambda x.\ M \to_\beta \lambda x.\ M'}$
- $\dfrac{M \to_\beta M'}{MN \to_\beta M'N}$
- $\dfrac{N \to_\beta N'}{MN \to_\beta MN'}$
- $\dfrac{M =_\alpha M' \qquad M' \to_\beta N' \qquad N' =_\alpha N}{M' \to_\beta N'}$

An example of this is as follows, note how it all ends up at yy - **confluence**;



We denote multi-step β-reduction as $\to_\beta^*$.

- $\dfrac{M =_\alpha M'}{M \to_\beta^* M'}$ reflexivity, α-conversion
- $\dfrac{M \to_\beta M'' \qquad M'' \to_\beta M'}{M \to_\beta^* M'}$ transitivity

The concept of **confluence** can be formally written as the following theorem (Church-Rosser);

$$\forall M, M_1, M_2.\ M \to_\beta^* M_1 \wedge M \to_\beta^* M_2 \Rightarrow \exists M'.\ M_1 \to_\beta^* M' \wedge M_2 \to_\beta^* M'$$

If you keep reducing a redex, you can eventually reach the same point, as seen above.

λ-terms are in β-normal form if they contain no redexes.

$$\text{is\_in\_nf}(M) \triangleq \forall M'.\ M \not\to_\beta M'$$

$$\text{has\_nf}(M) \triangleq \exists M'.\ M \to_\beta^* M' \wedge \text{is\_in\_nf}(M')$$

β-normal forms are unique;

$$\forall M, N_1, N_2.\ M \to_\beta^* N_1 \wedge M \to_\beta^* N_2 \wedge \text{is\_in\_nf}(N_1) \wedge \text{is\_in\_nf}(N_1) \Rightarrow N_1 =_\alpha N_2$$

The proof of this can be done with Church-Rosser. Obtain such an N such that both $N_1 \to_\beta^* N$ and $N_2 \to_\beta^* N$. However, since N_1 is in β-normal form, and $N_1 \to_\beta^* N$, it follows that $N_1 =_\alpha N$. The same can be said for N_2, and therefore we obtain $N_2 =_\alpha N$. As we have both these equivalences, we end up with $N_1 =_\alpha N =_\alpha N_2$.

β-equivalence

We can define $=_\beta$ as the smallest equivalence relation containing $\to_\beta$, or as $\to_\beta^*$ with symmetry. This can be formally defined as the following;

$$M_1 =_\beta M_2 \Leftrightarrow \exists M'.\ M_1 \to_\beta^* M' \wedge M_2 \to_\beta^* M'$$

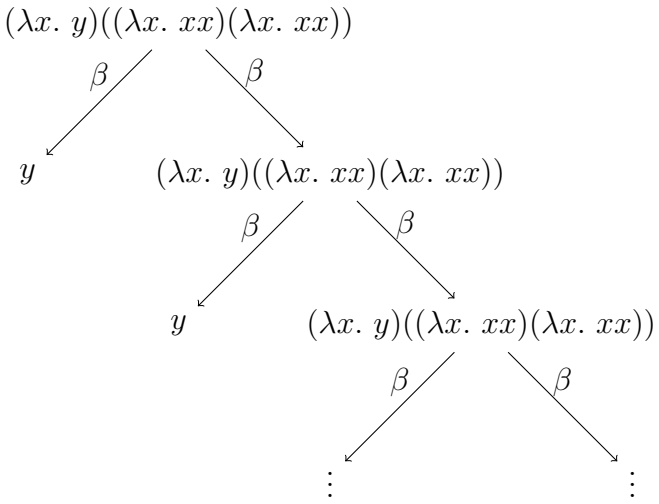
We can say two λ-terms are β-equivalent if we can find some λ-term such that both terms multi-step reduce to it. All the λ-terms in the diagram drawn in the previous lecture are β-equivalent.

Normalisation

- not all λ-terms have a normal form, for example

  $(\lambda x.\ xx)(\lambda x.\ xx) \to_\beta (\lambda x.\ xx)(\lambda x.\ xx) \to_\beta \cdots$

- the order of reduction matters, for example



Reduction Strategies

- **normal order**
  - evaluate the leftmost redex not contained in another redex
  - always reduces to a normal form (if one exists)
  - can perform computation in unevaluated function bodies (passes unevaluated functions as function arguments)

$$\begin{aligned}
&((\lambda xy.\ xyx)tu)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ &((\lambda y.\ tyt)u)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ &(tut)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ &(tut)((\lambda yz.\ v((\lambda x.\ xx)y))((\lambda x.\ xy)w)) \\
\to_\beta\ &(tut)(\lambda z.\ v((\lambda x.\ xx)((\lambda x.\ xy)w))) \\
\to_\beta\ &(tut)(\lambda z.\ v((\lambda x.\ xy)w)((\lambda x.\ xy)w))) \\
\to_\beta\ &(tut)(\lambda z.\ v((wy)((\lambda x.\ xy)w))) \\
\to_\beta\ &(tut)(\lambda z.\ v((wy)(wy)))
\end{aligned}$$

- **call by name**

- do not reduce under λ (do not reduce inside function bodies that haven't been fully evaluated), and do not evaluate function arguments until you have to
- does not always reduce a term to its normal form
- used in *Haskell* etc.

$$
\begin{aligned}
& ((\lambda xy.\ xyx)tu)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ & ((\lambda y.\ tyt)u)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ & (tut)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ & (tut)((\lambda yz.\ v((\lambda x.\ xx)y))((\lambda x.\ xy)w)) \\
\to_\beta\ & (tut)(\lambda z.\ v((\lambda x.\ xx)((\lambda x.\ xy)w)))
\end{aligned}
$$

- **call by value**
  - do not reduce under λ, and fully evaluate function arguments before plugging them in
  - finds normal forms less often
  - most popular strategy, evaluates function arguments only once

$$
\begin{aligned}
& ((\lambda xy.\ xyx)tu)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ & ((\lambda y.\ tyt)u)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ & (tut)((\lambda xyz.\ x((\lambda x.\ xx)y))v((\lambda x.\ xy)w)) \\
\to_\beta\ & (tut)((\lambda xyz.\ x((\lambda x.\ xx)y))v(wy)) \\
\to_\beta\ & (tut)((\lambda yz.\ v((\lambda x.\ xx)y))(wy)) \\
\to_\beta\ & (tut)(\lambda z.\ v((\lambda x.\ xx)(wy)))
\end{aligned}
$$

4th December 2019 — Lecture

Definability

A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ (takes n arguments) is λ-definable iff there exists a closed λ-term (no free variables) M with the following properties ($\underline{n}$ is the encoding of the natural number n in the λ-calculus);

$$f(x_1, \ldots, x_n) = y \Leftrightarrow M\underline{x_1} \ldots \underline{x_n} =_\beta y \text{ and } f(x_1, \ldots, x_n) \uparrow \Leftrightarrow M\underline{x_1} \ldots \underline{x_n} \text{ has no normal form}$$

The Church-Turing Thesis states that if f is computable iff f is register-machine-computable, Turing-machine-computable, or λ-definable, which means that these models of computation are equivalent.

Constructors

Note that the constructors are written in the syntax of *Coq*. An `Inductive` type can have recursion.

- Booleans `true, false`

```
Inductive Bool : Set =
  | true  : Bool
  | false : Bool
```

  Very simple type, with no underlying recursion - only two values.

- Pairs `(a, b)`

```
Inductive Pair : Set =
  | pair : elt -> elt -> Pair
```

Despite pairs containing arbitrary data, there is only one way of constructing pairs (`elt` means element). The notation is that `(a, b)` means `pair a b`.

- Lists `[e1, ..., en]`

```
1 Inductive List : Set
2  | nil  : List
3  | cons : elt -> List -> List
```

A list can either be empty (`nil`), or be a head element `elt` with a tail `List`. This is very similar to *Haskell*, so no further explanation **should** be needed.

- Binary trees I'm not drawing this out.

```
1 Inductive Tree : Set =
2  | leaf   : elt -> Tree
3  | branch : Tree -> elt -> Tree -> Tree
```

A binary tree is either a single `leaf`, or a `branch` which takes in the left subtree, an element, and the right subtree.

- Natural numbers `0, 1, 2, ...`

```
1 Inductive Nat : Set =
2  | z : Nat
3  | S : Nat -> Nat
```

This is Peano arithmetic, see **CO142**. The notation is that `z` is 0, `S z` is 1, `S (S z)` is 2, and so on.

Encoding Values

1. all values are λ-abstractions, one λ per constructor
2. the abstraction body is the formal definition of the value
3. any non-inductive parameters have to be encoded themselves

- Booleans

First shorten constructors to t and f, giving us a λ per constructor. The abstraction bodies are as follows;

$$\underline{\texttt{true}} = \lambda tf.\ t$$
$$\underline{\texttt{false}} = \lambda tf.\ f$$

- Pairs

First shorten constructor to p, giving us a single λ. The abstraction body is as follows;

$$\underline{\texttt{pair a b}} = \lambda p.\ p\ \underline{a}\ \underline{b}$$

- Lists

First shorten constructors to n and c, giving us a λ per constructor.

$$\underline{\texttt{cons x (cons y (cons z nil))}} = \lambda cn.\ c\ \underline{x}\ (c\ \underline{y}\ (c\ \underline{z}\ n)$$

- Binary trees

Same method as previously, etc.

$$\underline{\texttt{branch (leaf 1) 0 (leaf 2)}} = \lambda bl.\ b\ (l\ \underline{1})\ \underline{0}\ (l\ \underline{2})$$

Unpacking

Unpacking is the process of using application to remove the λs from an encoded value to get the body of the abstraction;

$$(\lambda tf.\ t)\ t\ f \to^*_\beta t$$
$$(\lambda p.\ p\ \underline{a}\ \underline{b})\ p \to^*_\beta p\ \underline{a}\ \underline{b}$$
$$(\lambda cn.\ c\ \underline{x}\ (c\ \underline{y}\ (c\ \underline{z}\ n)))\ c\ n \to^*_\beta c\ \underline{x}\ (c\ \underline{y}\ (c\ \underline{z}\ n))$$
$$(\lambda bl.\ b\ (l\ \underline{1})\ \underline{0}\ (l\ \underline{2}))\ b\ l \to^*_\beta b\ (l\ \underline{1})\ \underline{0}\ (l\ \underline{2})$$

Encoding Constructors

1. one λ per constructor
2. prepend any parameter λs to the constructor λs
3. abstraction body follows the constructor type, with all inductive parameters unpacked

$T = \lambda tf.\ t$	`true`
$F = \lambda tf.\ f$	`false`
$P = \lambda abp.\ p\ a\ b$	`pair`
$N = \lambda cn.\ n$	`nil`
$C = \lambda elcn.\ c\ e\ (l\ c\ n)$	`cons`

note how the `cons` constructor has $(l\ c\ n)$, as the inductive parameter must be unpacked

$L = \lambda ebl.\ l\ e$	`leaf`
$B = \lambda t_1 e t_2 bl.\ b\ (t_1\ b\ l)\ e\ (t_2\ b\ l)$	`branch`

A proof of this working for `cons` is as follows (note the use of $\to^*_\beta$ to denote multiple steps - important for exam);

$$\begin{aligned}
& C\ \underline{x}\ (C\ \underline{y}\ N) \\
\triangleq\ & C\ \underline{x}\ ((\lambda elcn.\ c\ e\ (l\ c\ n))\ \underline{y}\ N) \\
\to^*_\beta\ & C\ \underline{x}\ (\lambda cn.\ c\ \underline{y}\ (N\ c\ n)) \\
\triangleq\ & C\ \underline{x}\ (\lambda cn.\ c\ \underline{y}\ ((\lambda cn.\ n)\ c\ n)) \\
\to^*_\beta\ & C\ \underline{x}\ (\lambda cn.\ c\ \underline{y}\ n) \\
\triangleq\ & (\lambda elcn.\ c\ e\ (l\ c\ n))\ \underline{x}\ (\lambda cn.\ c\ \underline{y}\ n) \\
\to^*_\beta\ & (\lambda cn.\ c\ \underline{x}\ ((\lambda cn.\ c\ \underline{y}\ n)\ c\ n)) \\
\to^*_\beta\ & \lambda cn.\ c\ \underline{x}\ (c\ \underline{y}\ n)
\end{aligned}$$

Encoding Operations

The idea is to manipulate the encoding body by inserting a function in place of the constructor. For example, with a pair $\lambda p.\ p\ a\ b$, if we want the first element, we want a, otherwise we want b if we want the second element.

$$Fst = \lambda p.\ p\ (\lambda ab.\ a)$$
$$Snd = \lambda p.\ p\ (\lambda ab.\ b)$$

For example;

$$\begin{aligned}
& Fst(\lambda p.\ p\ \underline{0}\ \underline{1}) \\
\triangleq\ & (\lambda p.\ p\ (\lambda ab.\ a))(\lambda p.\ p\ \underline{0}\ \underline{1}) \\
\rightarrow_\beta\ & (\lambda p.\ p\ \underline{0}\ \underline{1})(\lambda ab.\ a) \\
\rightarrow_\beta\ & (\lambda ab.\ a)\ \underline{0}\ \underline{1} \\
\rightarrow_\beta^*\ & \underline{0}
\end{aligned}$$

Similarly for the head and tail of a list, we have;

$$\begin{aligned}
Hd &= \lambda l.\ l\ (\lambda el.\ e)\ _ \\
Tl &= \lambda l.\ l\ (\lambda el.\ (\lambda cn.\ l))\ _
\end{aligned}$$

Note the use of the $\_$, since the constructor for a list has two parameters, we want to discard the n (`nil`) constructor, since it doesn't make sense to use this on the empty list. It's also important to note that we also need the additional cn parameters for the tail to pack the list. However, now we have $Hd\ N =_\beta Tl\ N =_\beta \_$, since both those functions are total (in terms of λ-definability). Therefore this must be adjusted to not have a normal form.

Constructor Recognisers

In order to do this, we have to employ constructor recognisers, which are mechanisms designed to recognise the lead constructor then branch using that mechanism. A use case for this is for Hd to recognise whether it's being applied to the empty list (and thus diverge), or to proceed otherwise.

$$\begin{aligned}
isNil &= \lambda l.\ l\ (\lambda el.\ \underline{false})\ (\underline{true}) \\
isCons &= \lambda l.\ l\ (\lambda el.\ \underline{true})\ (\underline{false})
\end{aligned}$$

this can now be used for branching as follows;

$$ifNil = \lambda xyl.\ l\ (\lambda el.\ y)\ x$$

give it some term without a normal form (diverges) as follows;

$$D = (\lambda x.\ xx)(\lambda x.\ xx)$$

we can now create partial functions that can diverge (which is the correct use)

$$\begin{aligned}
PHd &= ifNil\ D\ Hd \\
PTl &= ifNil\ D\ Tl
\end{aligned}$$

Encoding Natural Numbers

The Church numerals are encoded as follows;

$$\begin{aligned}
\underline{0} &= \lambda sz.\ z \\
\underline{1} &= \lambda sz.\ s\ z \\
\underline{2} &= \lambda sz.\ s\ (s\ z) \\
\underline{3} &= \lambda sz.\ s\ (s\ (s\ z)) \\
&\dots \\
\underline{n} &= \lambda sz.\ \underbrace{s\ (\dots\ (s}_{n}\ z)\ \dots\ )
\end{aligned}$$

In order to encode addition of two natural numbers, m and n, we want;

$$\underline{m} = \lambda sz.\ \underbrace{s\ (\ldots(s}_{m}\ z)\ldots)$$
$$\underline{n} = \lambda sz.\ \underbrace{s\ (\ldots(s}_{n}\ z)\ldots)$$
$$plus\ \underline{m}\ \underline{n} =_\beta \lambda sz.\ \underbrace{s\ (\ldots(s}_{m+n}\ z)\ldots)$$

Ideally, we want to substitute the blue part of $\underline{n}$ into the red part of $\underline{m}$. To achieve this, we can do the following steps;

1. unpack to obtain the body of $\underline{n}$ — $\underline{n}\ s\ z =_\beta \underbrace{s(\ldots(s}_{n}\ z)\ldots)$
2. unpack $\underline{m}$ with the unpacked $\underline{n}$ in place of z — $\underline{m}\ s\ (\underline{n}\ s\ z) =_\beta \underbrace{s(\ldots(s}_{m+n}\ z)\ldots)$
3. pack into a Church numeral — $\lambda sz.\ \underline{m}\ s\ (\underline{n}\ s\ z) =_\beta \underline{m+n}$
4. make this into a function that takes m and n — $plus \triangleq \lambda mnsz.\ m\ s\ (n\ s\ z)$

5th December 2019 — Lecture

Multiplication

For short, let us define n applications of s as follows; s^n, allowing for the following;

$$\underline{m} = \lambda sz.\ s^m\ z$$
$$\underline{n} = \lambda sz.\ s^n\ z$$
$$\underline{m+n} = \lambda sz.\ s^{m+n}\ z$$

To encode multiplication, we only partially unpack $\underline{n}$, using $(n\ s)$, instead of $(n\ s\ z)$ in place of s in $\underline{m}$ (similar strategy to what we did for addition).

$$\begin{aligned}
m\ (n\ s) &= (\lambda sz.\ s^m\ z)(n\ s) \\
&= (\lambda sz.\ s^m\ z)(\lambda z.\ s^n\ z) \\
&=_\beta (\lambda z.\ (\lambda z.\ s^n\ z)^m\ z) \\
&\to_\beta (\lambda z.\ (\lambda z.\ s^n\ z)^{m-1}\ (s^n\ z)) \\
&\to_\beta (\lambda z.\ (\lambda z.\ s^n\ z)^{m-2}\ (s^{2\cdot n}\ z)) \\
&\to_\beta^* (\lambda z.\ s^{m\cdot n}\ z) \\
\underline{m\cdot n} &= \lambda sz.\ s^{m\cdot n}\ z \\
&= \lambda s.\ m\ (n\ s) \\
mult &\triangleq \lambda mns.\ m\ (n\ s)
\end{aligned}$$

Combinators

Combinators are closed λ-terms. The most important ones are I, K, and S;

$I \triangleq \lambda x.\ x$ — identity

$I : \alpha \to \alpha$

$K \triangleq \lambda xy.\ x$ — true / selector

$K : \alpha \to \beta \to \alpha$

$S \triangleq \lambda xyz.\ xz(yz)$

$S : \underbrace{(\alpha \to \beta \to \gamma)}_{x} \to \underbrace{(\alpha \to \beta)}_{y} \to \underbrace{\alpha}_{z} \to \gamma$

All of the λ-calculus we have covered can be done with the SKI combinators (I is also SKK).

Recursion

For an example of recursion, we will be using the factorial.

$$
\begin{aligned}
fact\ n &\triangleq \text{if } (n = 0) \text{ then } 1 \text{ else } n \cdot fact\ (n-1) \\
ifz\ \underline{n}\ x\ y &=_\beta \begin{cases} x & n = 0 \\ y & n > 0 \end{cases} \\
fact &=_\beta \lambda n.\ ifz\ n\ \underline{1}\ (mult\ n\ (fact\ (pred\ n))) \\
&=_\beta \underbrace{(\lambda f.\ \lambda n.\ ifz\ n\ \underline{1}\ (mult\ n\ (f\ (pred\ n))))}_{F}\ fact
\end{aligned}
$$

In mathematics, if $f(x) = x$, then x is a **fixpoint** of f. As such, we have $fact$ being a fixpoint of F.

$$
\begin{aligned}
Y &\triangleq \lambda f.\ (\lambda x.\ f\ (xx))(\lambda x.\ f\ (xx)) \\
Y\ f &\triangleq (\lambda f.\ (\lambda x.\ f\ (xx))(\lambda x.\ f\ (xx)))\ f \\
&\to_\beta (\lambda x.\ f\ (xx))(\lambda x.\ f\ (xx)) \\
&\to_\beta f\ ((\lambda x.\ f\ (xx))\ (\lambda x.\ f\ (xx))) \\
&=_\beta f\ (Y\ f)
\end{aligned}
$$

Therefore, $Y\ f$ is a fixpoint of f. The Y combinator is the **fixpoint operator**. As such, we can define the factorial as;

$$fact \triangleq F\ (\lambda f.\ \lambda n.\ ifz\ n\ \underline{1}\ (mult\ n\ (f\ (pred\ n))))$$

η-equivalence

$(\lambda x.\ f\ x) \neq_\beta f$, despite always having the case $(\lambda x.\ f\ x)\ M =_\beta f\ M$. However, there is no way to capture this with β-reduction. This can be captured with the following rule;

$$\frac{x \notin \mathrm{FV}(M)}{\lambda x.\ M\ x =_\eta M}$$