

CO240 - Models of Computation

9th October 2019

Lecture

Hilbert's Entscheidungsproblem

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.

Entscheidungsproblem 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 Entscheidungsproblem, 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

```

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

```

```

1 power' x n
2   | n == 0 = 1
3   | even n = k^2
4   | odd n  = x*k^2
5   where
6     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.

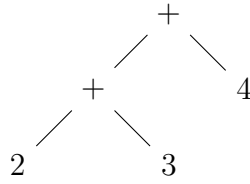
$$\begin{aligned}
 B \in \text{Bool} &::= \text{true} \mid \text{false} \mid E = E \mid E < E \mid B \& B \mid \neg B \mid \dots \\
 E \in \text{Exp} &::= x \mid n \mid E + E \mid \dots \\
 C \in \text{Com} &::= x := E \mid \text{if } B \text{ then } C \text{ else } C \mid C; C \mid \text{skip} \mid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

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.

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

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 $E \rightarrow \dots \rightarrow n$ (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)

- (B-ADD) $n_3 = n_1 + n_2$

$$\frac{\overline{n \Downarrow n} \quad \frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}}{E_1 + E_2 \Downarrow n_3}$$

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

$$\frac{3 \Downarrow 3 \quad \frac{2 \Downarrow 2 \quad 1 \Downarrow 1}{2 + 1 \Downarrow 3}}{3 + (2 + 1) \Downarrow 6}$$

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)

$$\frac{E_1 \rightarrow E'_1}{E_1 + E_2 \rightarrow E'_1 + E_2}$$

- (S-RIGHT)

$$\frac{E \rightarrow E'}{n + E \rightarrow n + E'}$$

- (S-ADD) $n_3 = n_1 + n_2$

$$\frac{}{n_1 + n_2 \rightarrow n_3}$$

For example, consider the small-step evaluation of;

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

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

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

$$E \rightarrow E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_k \rightarrow E'$$

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

- **determinacy** $\forall E, E_1, E_2 [E \rightarrow E_1 \wedge E \rightarrow E_2 \Rightarrow E_1 = E_2]$
with big-step, it was with respect to numbers, but here it is with respect to all the small computational steps
- **confluence** $\forall E, E_1, E_2 [E \rightarrow^* E_1 \wedge E \rightarrow^* E_2 \Rightarrow \exists E' [E_1 \rightarrow^* E' \wedge E_2 \rightarrow^* E']]$
- **(strong) normalisation**
there are no infinite sequences of expressions, which means that any evaluation path will eventually reach a normal form
- **theorem** $\forall E, n_1, n_2 [E \rightarrow^* n_1 \wedge E \rightarrow^* n_2 \Rightarrow n_1 = n_2]$

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

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.

$$\frac{\frac{4 \Downarrow 4 \quad 1 \Downarrow 1}{(4 + 1) \Downarrow 5} \quad \frac{2 \Downarrow 2 \quad 2 \Downarrow 2}{(2 + 2) \Downarrow 4}}{(4 + 1) + (2 + 2) \Downarrow 9}$$

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

$$\frac{\frac{3 \Downarrow 3 \quad 2 \Downarrow 2}{(3 + 2) \Downarrow 5} \quad \frac{1 \Downarrow 1 \quad 4 \Downarrow 4}{(1 + 4) \Downarrow 5}}{(3 + 2) \times (1 + 4) \Downarrow 25}$$

3. Extending big-step semantics for subtraction;

$$\frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_3 \Downarrow n_3}$$

$$\frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_3 \Downarrow n_3}$$

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

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 $(5 - 7) + 10 = 10$

- $= \text{NaN}$

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;

$$\frac{E_1 \Downarrow \text{NaN} \quad E_2 \Downarrow n}{E_1 \pm E_2 \Downarrow \text{NaN}}$$

- extend it to all numbers

4. Sound was broken for this part on Panopto.

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

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

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 $(1 + 2) \ ? \ 4 \Downarrow 3$, and also the outcome $(1 + 2) \ ? \ 4 \Downarrow 4$;

$$\frac{E_1 \Downarrow n_1}{E_1 \ ? \ E_2 \Downarrow n_1}$$

$$\frac{E_2 \Downarrow n_2}{E_1 \ ? \ E_2 \Downarrow n_2}$$

(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, $E_1 \ ? \ E_2$ 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;

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

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)
$$\frac{\langle E_1, s \rangle \rightarrow_e \langle E'_1, s' \rangle}{\langle E_1 + E_2, s \rangle \rightarrow_e \langle E'_1 + E_2, s' \rangle}$$
- (W-EXP.RIGHT)
$$\frac{\langle E, s \rangle \rightarrow_e \langle E', s' \rangle}{\langle n + E, s \rangle \rightarrow_e \langle n + E', s' \rangle}$$
- (W-EXP.ADD)
$$n_3 = n_1 + n_2 \quad \frac{}{\langle n_1 + n_2, s \rangle \rightarrow_e \langle n_3, s \rangle}$$
- (W-EXP.VAR)
$$s(x) = n \quad \frac{}{\langle x, s \rangle \rightarrow_e \langle n, s \rangle}$$

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;

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

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

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

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 ($n_1 = n_2$)

$$\frac{n_1 = n_2}{\langle n_1 = n_2, s \rangle \rightarrow_b \langle \text{true}, s \rangle}$$

$$\frac{n_1 \neq n_2}{\langle n_1 = n_2, s \rangle \rightarrow_b \langle \text{false}, s \rangle}$$

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)

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

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)
$$\frac{}{\langle \text{skip}; C_2, s \rangle \rightarrow_c \langle C_2, s \rangle}$$

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

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

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

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

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;

$$\frac{\frac{\frac{\langle x, s \rangle \rightarrow_e \langle 5, s \rangle}{\langle z := x, s \rangle \rightarrow_c \langle z := 5, s \rangle}}{\langle z := x; x := y, s \rangle \rightarrow_c \langle z := 5; x := y, s \rangle}}{\langle (z := x; x := y); y := z, s \rangle \rightarrow_c \langle (z := 5; x := y); y := z, s \rangle}$$

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;

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

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 $x := (x++) + (x++)$ as such;

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

17th October 2019

Tutorial

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

- LHS evaluated first
$$\frac{\langle C_1, s \rangle \rightarrow_c \langle C'_1, s' \rangle}{\langle C_1 \parallel C_2, s \rangle \rightarrow_c \langle C'_1 \parallel C_2, s' \rangle}$$
- RHS evaluated first
$$\frac{\langle C_2, s \rangle \rightarrow_c \langle C'_2, s' \rangle}{\langle C_1 \parallel C_2, s \rangle \rightarrow_c \langle C_1 \parallel C'_2, s' \rangle}$$
- Collapsing after both sides are complete
$$\frac{}{\langle \text{skip} \parallel \text{skip}, s \rangle \rightarrow_c \langle \text{skip}, s \rangle}$$

Note that this is an example of non-determinism, as we can see with the following example. Consider the program $(x := 1) \parallel (x := 2; x := x + 2)$, 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 $\langle C, s \rangle \rightarrow_c^* \langle \text{skip}, s' \rangle$. Let $C =$
`while true do skip;`

$$\begin{aligned} & \langle \text{while true do skip}, s \rangle \rightarrow \\ & \langle \text{if true (skip; while true do skip)}, s \rangle \rightarrow_c \\ & \langle \text{skip; while true do skip}, s \rangle \rightarrow_c \\ & \langle \text{while true do skip}, s \rangle \end{aligned}$$

Now let us assume that there is some finite evaluation path that takes n steps, such that $\langle C, s \rangle \rightarrow^* \langle \text{skip}, s' \rangle$, 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**).

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

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.

Determinancy