CO240 - Models of Computation

9th October 2019 Lecture

Hilbert's Entscheidungproblem

Is there an algorithm, when fed any statement in the formal laguage 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. Algorthms 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 usch 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 gies 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 algorhtms 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 assigment to a variable.

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

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.

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

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.

$$\begin{tikzcd}
 & & + \arrow[dl, dash] \arrow[dr, dash] & \\
 & + \arrow[dl, dash] \arrow[dr, dash] & & 4 \\
2 & & 3 &
\end{tikzcd}$$

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 $E \to \ldots \to n$ (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) $n_3 = n_1 + n_2$ $$\frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}$$

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 $\Downarrow$;

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

  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;

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

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 $\rightarrow$;

- **determinancy** $\forall E, E_1, E_2[E \rightarrow E_1 \land E \rightarrow E_2 \Rightarrow E_1 = E_2]$

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

  tehre are no infinite sequences of expressions, which means that any evaluation path will evntually reach a normal form
- **theorem** $\forall E, n_1, n_2[E \rightarrow^* n_1 \land E \rightarrow^* n_2 \rightarrow n_1 = n_2]$

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