Lecture
Models of Computation
(DIT310, TDA184)

Nils Anders Danielsson

2016-12-12

Today

- Repetition. Please interrupt if you want to discuss something in more detail.
- Course evaluation.

Models of computation

- Actual hardware or programming languages: Lots of (irrelevant?) details.
- In this course: Idealised models of computation.
- PRF, RF.
- X.
- Turing machines.

The Church-Turing thesis

- The thesis:
  Every effectively calculable function on the positive integers can be computed using a Turing machine.
- Widely believed to be true.
- Many models are Turing-complete.

Comparing sets' sizes

- Injections, surjections, bijections.
- Countable (injection to $\mathbb{N}$), uncountable.
- Diagonalisation.
- Not every function is computable.

Inductively defined sets

An inductively defined set:

$$\frac{}{\mathsf{nil} \in \mathit{List}\ A} \qquad \frac{x \in A \qquad xs \in \mathit{List}\ A}{\mathsf{cons}\ x\ xs \in \mathit{List}\ A}$$

Primitive recursion:

$$\begin{aligned}
&\mathit{listrec} \in B \to (A \to \mathit{List}\ A \to B \to B) \to \\
&\qquad\qquad \mathit{List}\ A \to B \\
&\mathit{listrec}\ n\ c\ \mathsf{nil} \qquad\qquad\quad = n \\
&\mathit{listrec}\ n\ c\ (\mathsf{cons}\ x\ xs) = c\ x\ xs\ (\mathit{listrec}\ n\ c\ xs)
\end{aligned}$$

Inductively defined sets

An inductively defined set:

$$\frac{}{\mathsf{nil} \in \mathit{List}\ A} \qquad \frac{x \in A \qquad xs \in \mathit{List}\ A}{\mathsf{cons}\ x\ xs \in \mathit{List}\ A}$$

Structural induction (P: a predicate on $\mathit{List}\ A$):

$$\frac{\begin{array}{c} P\ \mathsf{nil} \\ \forall x \in A.\ \forall\ xs \in \mathit{List}\ A.\ P\ xs \Rightarrow P\ (\mathsf{cons}\ x\ xs) \end{array}}{\forall xs \in \mathit{List}\ A.\ P\ xs}$$

Quiz

Write down the type of one of the higher-order primitive recursion schemes for the following inductively defined set:

$$\frac{n \in \mathbb{N}}{\textsf{leaf}\ n \in \mathit{Tree}} \qquad \frac{l, r \in \mathit{Tree}}{\textsf{node}\ l\ r \in \mathit{Tree}}$$

PRF

Sketch:

$$
\begin{aligned}
f\ () &= \mathsf{zero} \\
f\ (x) &= \mathsf{suc}\ x \\
f\ (x_1, \ldots, x_k, \ldots, x_n) &= x_k \\
f\ (x_1, \ldots, x_n) &= g\ (h_1\ (x_1, \ldots, x_n), \ldots, h_k\ (x_1, \ldots, x_n))
\end{aligned}
$$

$$
\begin{aligned}
f\ (x_1, \ldots, x_n, \mathsf{zero}) &= g\ (x_1, \ldots, x_n) \\
f\ (x_1, \ldots, x_n, \mathsf{suc}\ x) &= \\
&\quad h\ (x_1, \ldots, x_n, f\ (x_1, \ldots, x_n, x), x)
\end{aligned}
$$

PRF

- Abstract syntax (PRF_n).
- Denotational semantics:

$$[\![\_]\!] \in PRF_n \to (\mathbb{N}^n \to \mathbb{N})$$

- Big-step operational semantics:

$$f\ [\rho]\ \Downarrow\ n$$

PRF

- Strictly weaker than χ/Turing machines.
- Some χ-computable *total* functions are not PRF-computable.
- This is the case for any model of computation where all programs "terminate", given certain assumptions.

Not exactly the χ-computable functions

Assumptions:

- Programs: $Prog$.
- Total, χ-computable semantics:

  $$[\![\_]\!] \in Prog \times \mathbb{N} \to \mathbb{N}$$

- A coding function:

  $$code \in Prog \to \mathbb{N}$$

- A χ-computable left inverse of $code$:

  $$decode \in \mathbb{N} \to Prog$$

Not exactly the χ-computable functions

- Define $g \in \mathbb{N} \to \mathbb{N}$ by

  $$g\ n = [\![(decode\ n, n)]\!] + 1.$$

  Note that g is total and χ-computable.
- Assume that $\underline{g} \in Prog$, with

  $$\forall\ n \in \mathbb{N}.\ [\![(\underline{g}, n)]\!] = g\ n.$$

- We get a contradiction:

  $$\begin{aligned} & g\ (code\ \underline{g}) && = \\ & [\![(decode\ (code\ \underline{g}), code\ \underline{g})]\!] + 1 && = \\ & [\![(\underline{g}, code\ \underline{g})]\!] + 1 && = \\ & g\ (code\ \underline{g}) + 1 \end{aligned}$$

RF

- PRF $+$ minimisation.
- For $f \in \mathbb{N} \rightharpoonup \mathbb{N}$:
  f is RF-computable $\Leftrightarrow$
  f is χ-computable $\Leftrightarrow$
  f is Turing-computable.

X

$$
\begin{aligned}
e ::= {} & x \\
\mid {} & (e_1\ e_2) \\
\mid {} & \lambda x.\ e \\
\mid {} & \mathsf{C}(e_1, \ldots, e_n) \\
\mid {} & \textbf{case}\ e\ \textbf{of}\ \{\, \mathsf{C}_1(x_1, \ldots, x_n) \rightarrow e_1; \ldots \} \\
\mid {} & \textbf{rec}\ x = e
\end{aligned}
$$

- Untyped, strict.
- $\textbf{rec}\ x = e \;\approx\; \textbf{let}\ x = e\ \textbf{in}\ x$.

X

- Abstract syntax.
- Substitution of closed expressions.
- Big-step operational semantics, not total.
- The semantics as a partial function:

$$\llbracket \_ \rrbracket \in \mathit{CExp} \rightharpoonup \mathit{CExp}$$

- Representing inductively defined sets.

Representing expressions

Coding function:

$$
\begin{aligned}
\ulcorner \_ \urcorner &\in \mathit{Exp} \rightarrow \mathit{CExp} \\
\ulcorner x \urcorner &= \mathsf{Var}(\ulcorner x \urcorner) \\
\ulcorner e_1\ e_2 \urcorner &= \mathsf{Apply}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner) \\
\ulcorner \lambda x.\ e \urcorner &= \mathsf{Lambda}(\ulcorner x \urcorner, \ulcorner e \urcorner) \\
&\vdots
\end{aligned}
$$

Representing expressions

Coding function:

$$
\begin{aligned}
&\ulcorner \_ \urcorner \in \mathit{Exp} \to \mathit{CExp} \\
&\ulcorner \mathsf{var}\ x \urcorner &&= \mathsf{const}\ \ulcorner \mathsf{Var} \urcorner\ (\mathsf{cons}\ \ulcorner x \urcorner\ \mathsf{nil}) \\
&\ulcorner \mathsf{apply}\ e_1\ e_2 \urcorner &&= \mathsf{const}\ \ulcorner \mathsf{Apply} \urcorner \\
& &&\qquad (\mathsf{cons}\ \ulcorner e_1 \urcorner\ (\mathsf{cons}\ \ulcorner e_2 \urcorner\ \mathsf{nil})) \\
&\ulcorner \mathsf{lambda}\ x\ e \urcorner &&= \mathsf{const}\ \ulcorner \mathsf{Lambda} \urcorner \\
& &&\qquad (\mathsf{cons}\ \ulcorner x \urcorner\ (\mathsf{cons}\ \ulcorner e \urcorner\ \mathsf{nil})) \\
&\vdots
\end{aligned}
$$

Representing expressions

Coding function:

$$
\begin{aligned}
&\ulcorner \_ \urcorner \in \mathit{Exp} \to \mathit{CExp} \\
&\ulcorner \textsf{var}\ x \urcorner &&= \textsf{const}\ \ulcorner \textsf{Var} \urcorner\ (\textsf{cons}\ \ulcorner x \urcorner\ \textsf{nil}) \\
&\ulcorner \textsf{apply}\ e_1\ e_2 \urcorner &&= \textsf{const}\ \ulcorner \textsf{Apply} \urcorner \\
& &&\qquad (\textsf{cons}\ \ulcorner e_1 \urcorner\ (\textsf{cons}\ \ulcorner e_2 \urcorner\ \textsf{nil})) \\
&\ulcorner \textsf{lambda}\ x\ e \urcorner &&= \textsf{const}\ \ulcorner \textsf{Lambda} \urcorner \\
& &&\qquad (\textsf{cons}\ \ulcorner x \urcorner\ (\textsf{cons}\ \ulcorner e \urcorner\ \textsf{nil})) \\
&\vdots
\end{aligned}
$$

Alternative type:

$$\ulcorner \_ \urcorner \in \mathit{Exp}\ A \to \mathit{CExp}\ (\mathit{Rep}\ A)$$

$\mathit{Rep}\ A$: Representations of programs of type A.

Computability

- $f \in A \rightharpoonup B$ is χ-computable if

$$\exists\, e \in \mathit{CExp}.\ \forall\, a \in A.\ [\![e \ulcorner a \urcorner]\!] = \ulcorner f\ a \urcorner.$$

- Use reasonable coding functions:
  - Injective.
  - Computable. But how is this defined?
- X-decidable: $f \in A \to \mathit{Bool}$.
- X-semi-decidable:
  If $f\ a =$ false then $[\![e \ulcorner a \urcorner]\!]$ is undefined.

Some computable partial functions

- The semantics $[\![\_]\!] \in \mathit{CExp} \rightharpoonup \mathit{CExp}$:

$$\forall\ e \in \mathit{CExp}.\ [\![\mathit{eval} \ulcorner e \urcorner]\!] = \ulcorner [\![e]\!] \urcorner.$$

- The coding function $\ulcorner \_ \urcorner \in \mathit{Exp} \to \mathit{CExp}$:

$$\forall\ e \in \mathit{Exp}.\ [\![\mathit{code} \ulcorner e \urcorner]\!] = \ulcorner \ulcorner e \urcorner \urcorner.$$

- The "Terminates in n steps?" function $\mathit{terminates\text{-}in} \in \mathit{CExp} \times \mathbb{N} \to \mathit{Bool}$:

$$\forall\ p \in \mathit{CExp} \times \mathbb{N}.$$
$$[\![\underline{\mathit{terminates\text{-}in}} \ulcorner p \urcorner]\!] = \ulcorner \mathit{terminates\text{-}in}\ p \urcorner.$$

Some non-computable partial functions

The halting problem with self-application,

$$
\begin{aligned}
&\mathit{halts\text{-}self} \in \mathit{CExp} \rightarrow \mathit{Bool}\\
&\mathit{halts\text{-}self}\ p =\\
&\quad \textbf{if}\ p \ulcorner p \urcorner\ \text{terminates}\ \textbf{then}\ \text{true}\ \textbf{else}\ \text{false},
\end{aligned}
$$

can be reduced to the halting problem,

$$
\begin{aligned}
&\mathit{halts} \in \mathit{CExp} \rightarrow \mathit{Bool}\\
&\mathit{halts}\ p = \textbf{if}\ p\ \text{terminates}\ \textbf{then}\ \text{true}\ \textbf{else}\ \text{false}.
\end{aligned}
$$

Some non-computable partial functions

Proof sketch:

- Assume that $\underline{halts}$ implements $halts$.
- Define $\underline{halts\text{-}self}$ in the following way:

$$\underline{halts\text{-}self} = \lambda\, p.\ \underline{halts}\ \mathsf{Apply}(p, code\ \ p)$$

- $\underline{halts\text{-}self}$ implements $halts\text{-}self$,

$$\begin{array}{l} \forall\ e \in CExp. \\ \quad [\![\underline{halts\text{-}self}\ \ulcorner e \urcorner]\!] = \ulcorner halts\text{-}self\ \ e \urcorner, \end{array}$$

because $\mathsf{Apply}(\ulcorner e \urcorner, code\ \ulcorner e \urcorner) \Downarrow \ulcorner e \ulcorner e \urcorner \urcorner$.

Some non-computable partial functions

The halting problem can be reduced to:

- Semantic equality:

$$\begin{aligned}
&\mathit{equal} \in \mathit{CExp} \times \mathit{CExp} \to \mathit{Bool} \\
&\mathit{equal}\ (e_1, e_2) = \\
&\quad \textbf{if } [\![e_1]\!] = [\![e_2]\!] \textbf{ then } \textsf{true} \textbf{ else } \textsf{false}
\end{aligned}$$

- Pointwise equality of elements in $\mathit{Fun} = \{ f \in \mathbb{N} \to \mathit{Bool} \mid f \text{ is } \chi\text{-computable} \}$:

$$\begin{aligned}
&\mathit{pointwise\text{-}equal} \in \mathit{Fun} \times \mathit{Fun} \to \mathit{Bool} \\
&\mathit{pointwise\text{-}equal}\ (f, g) = \\
&\quad \textbf{if } \forall\ n \in \mathbb{N}.\ f\ n = g\ n \textbf{ then } \textsf{true} \textbf{ else } \textsf{false}
\end{aligned}$$

Quiz

What is wrong with the following reduction of the halting problem to $\mathit{pointwise\text{-}equal}$?

$$
\begin{array}{l}
\underline{\mathit{halts}} = \lambda\, p.\ \underline{\mathit{not}}\ (\underline{\mathit{pointwise\text{-}equal}} \\
\quad \mathsf{Lambda}(\ulcorner\, n\, \urcorner, \\
\quad\quad \mathsf{Apply}(\ulcorner\, \underline{\mathit{terminates\text{-}in}}\, \urcorner, \\
\quad\quad\quad \mathsf{Const}(\ulcorner\, \mathsf{Pair}\, \urcorner, \\
\quad\quad\quad\quad \mathsf{Cons}(p, \mathsf{Cons}(\mathsf{Var}(\ulcorner\, n\, \urcorner), \mathsf{Nil}())))))) \\
\quad \ulcorner\, \lambda\, \_.\ \mathsf{False}()\, \urcorner)
\end{array}
$$

Bonus question: How can the problem be fixed?

Some non-computable partial functions

The halting problem can be reduced to:

- An optimal optimiser:

$$
\begin{aligned}
&\mathit{optimise} \in \mathit{CExp} \to \mathit{CExp} \\
&\mathit{optimise}\ e = \\
&\quad \text{some optimally small expression with} \\
&\quad \text{the same semantics as } e
\end{aligned}
$$

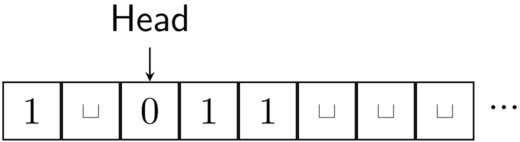
- Is a computable real number equal to zero?

$$
\begin{aligned}
&\mathit{is\text{-}zero} \in \mathit{Interval} \to \mathit{Bool} \\
&\mathit{is\text{-}zero}\ x = \mathbf{if}\ [\![x]\!] = 0\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false}
\end{aligned}
$$

- Many other functions, see Rice's theorem.

Turing machines

- A tape with a head:



- A state.
- Rules.

Turing machines

- Abstract syntax.
- Small-step operational semantics.
- The semantics as a family of partial functions:

$$[\![\_]\!] \in \ \forall\, tm \in TM.\ List\ \Sigma_{tm} \rightharpoonup List\ \Gamma_{tm}$$

- Several variants:
  - Accepting states.
  - Possibility to stay put.
  - A tape without a left end.
  - Multiple tapes.
  - Only two symbols (plus $\sqcup$).

Turing-computability

- Representing inductively defined sets.
- Turing-computable partial functions.
- Turing-decidable languages.
- Turing-recognisable languages.

Some computable partial functions

- The semantics (uncurried):

  $$\{\, (tm, xs) \mid tm \in TM, xs \in List\ \Sigma_{tm} \,\} \rightharpoonup List\ \Gamma_{tm}$$

  Self-interpreter/universal TM.
- The χ semantics.

Equivalence

- The Turing machine semantics is also χ-computable.
- Functions $f \in \mathbb{N} \rightharpoonup \mathbb{N}$ are Turing-computable iff they are χ-computable.

Finally

- We have studied the concept of "computation".
- How can "computation" be formalised?
  - To simplify our work: Idealised models.
  - The Church-Turing thesis.
- We have explored the limits of computation:
  - Programs that can run arbitrary programs.
  - A number of non-computable problems.

Good luck!