Notions of computability

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Are there limits to what we can compute?
- Are there limits to logical knowledge?
- Are these two related?

Introduction

Gödel numbering

- To work out the proofs of today we will build self-referential objects
- We need to concisely represent programs, statements, etc.
- We encode such objects as natural numbers

Introduction

Gödel numbering

- This means that any statement within our system is now a natural number
- "Properties of statements" $\Leftrightarrow$ "properties of numbers"
- There are many ways to define this encoding; we see Gödel's original

Introduction

Gödel numbering

- First assign a unique natural number to each basic symbol in the language
  - `if` $= 0$
  - `var` $= 1$
  - `,` $= 2$
  - ...

Introduction

Gödel numbering

- Given any formula $x_1, x_2, \ldots, x_n$, x_i is a number from the list above
- $enc(x_1, \ldots, x_n) = 2^{x_1} \cdot 3^{x_2} \cdot \cdots \cdot p_n^{x_n}$
- p_n is the n-th prime number
- According to the fundamental theorem of arithmetic, we can always extract back the original sequence

Introduction

Halting problem

- Consider the problem of determining whether a program terminates on input `i`
- We wish to find a function $h(i, x)$ that returns:
  - 1 if program i terminates on input x
  - 0 otherwise

Introduction

Halting problem

- We want to translate function $h(i, x)$ into a program, thus h needs to be:
  - computable (i.e. can be encoded in a Turing machine or equivalent)
  - total (i.e. for every pair of inputs it gives back a result)

Introduction

Halting problem

- We proceed by showing that no arbitrary total computable function f can be equal to h
- Consider partial computable function $g(i)$ that returns:
  - 0 if $f(i, i) = 0$
  - $\uparrow$ otherwise[a]

[a] $\uparrow$ is undefined return value, or infinite looping

Introduction

Halting problem

- Because f is totally computable, then also g is computable
- Thus we can give a program e_g that computes g and encode it with Gödel's numbering

Introduction

Halting problem

- $h(e, e)$ will always be different from $f(e, e)$
  - $f(e, e) = 0 \rightarrow g(e) = 0 \rightarrow h(e, e) = 1$
  - $f(e, e) = 1 \rightarrow g(e) = \uparrow \rightarrow h(e, e) = 0$

Introduction

Halting problem

- Function f was an arbitrary total computable function
- This means that there h is different from any (all) total computable functions
- Thus h is not **fully solvable** within a Turing machine
  - We may get approximated results, like `true`, `false`, `unknown`
  - We may get a partial function, that may sometimes loop forever
  - We will not get only a single (correct) `true`, `false` answer for any input

Introduction

Equivalent problems to the halting problem

- **EQUIVALENCE PROBLEM** Given two programs, test to see if they are equivalent
- **SIZE OPTIMIZATION PROBLEM** - Given a program, find the shortest equivalent program
- **GRAMMAR PROBLEM** Given two grammars, find out whether they define the same language
- ...

Introduction

Equivalent problems to the halting problem

- Equivalence problem reduction from the halting problem
  - To test whether program `P` halts on input `I`
  - Construct a new program `Q` like `P` but
    - always returning `true` whenever `P` returns something
  - Construct a trivial program `T` which always returns `true`
  - Find out whether `Q` and `T` are equivalent (on input `I`)
  - If so `Q` halts, otherwise it does not

Introduction

Equivalent problems to the halting problem

- Size optimization problem reduction from the halting problem
  - Construct a program that generated programs of increasing length
  - Run those that halt and compare their final output with the desired output
  - Take program that produces the desired output
- Shortest problem is *Kolmogorov complexity* of desired output

Introduction

But a computer is finite!

- Memory is finite! Haha!

Introduction

But a computer is finite!

- Memory is finite! Haha!
- A non-terminating program will eventually reach the same memory configuration
- We just need to check after 2^{10^9} steps to see if an old configuration is found
  - Also, we need to store all old states of the program somewhere
  - Potentially they are all as big as memory

Introduction

But a computer is finite!

- Now consider a program that is connected to the Internet
- We would need to store all states of all other connected machines

Introduction

But a computer is finite!

- Now consider a program that is connected to the Internet
- We would need to store all states of all other connected machines
- We really need a proper solution, this one is not practical

Introduction

Encoding dependence?

- Is there a preferred encoding which "unlocks" these problems and makes them computable?
- There are many "languages" for expressing computable functions
- Studied in the context of computability

Introduction

Encoding dependence?

- These are the "mathematical assembly languages" of theoretical Computer Science
- All can be implemented to modern architectures, and vice-versa
  - Turing machine
  - Von Neumann machine
  - Church's λ-calculus
  - ...

Introduction

Encoding dependence?

- Each of these formalisms can be translated into the other with an "interpreter"
- This means that anything one can do, the others can do as well
  - with the "overhead" of an interpreter

Introduction

Outside of programming?

- One might be tempted to look outside of programming
- Maybe mathematical encodings offer a solution?

Introduction

Outside of programming?

- Apparently we have the same issue
- Exemplified by Russel's paradox
- $R = \{x.x \notin x\}$
  - $R \in R$?
  - $R \notin R$?

Introduction

Gödel's incompleteness theorem

- A devastating generalization of Russel's paradox
- First incompleteness theorem:
  - no consistent system of axioms whose theorems can be listed by an "effective procedure"
  - capable of proving all truths about the relations of the natural numbers
  - For any such system, there will always be statements about the natural numbers that are true, but unprovable
- Proven by constructing self-referencing objects

Introduction

Curry-Howard isomorphism

- the direct relationship between computer programs and mathematical proofs
- observation that two families of formalisms that had seemed unrelated are in fact structurally the same kind of objects
- *a proof is a program, the formula it proves is a type for the program*
- *running a program is equivalent to "simplifying" a theorem*

Natural deduction isomorphism with λ-calculus

Intuitionistic implicational natural deduction	Lambda calculus type assignment rules
$\dfrac{}{\Gamma_1, \alpha, \Gamma_2 \vdash \alpha}\,\mathrm{Ax}$	$\dfrac{}{\Gamma_1, x : \alpha, \Gamma_2 \vdash x : \alpha}$
$\dfrac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta} \to I$	$\dfrac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}$
$\dfrac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta} \to E$	$\dfrac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash u : \alpha}{\Gamma \vdash t\ u : \beta}$

Conclusions

Conclusions

- Where does this leave us?

Conclusions

Conclusions

- Where does this leave us?
- Crying in a corner:
  - mathematics and informatics are isomorphic
  - mathematics and informatics are structurally unable of providing absolutely perfect answers to some questions
  - the choice of formalism, language, etc. does not change this
  - different formalisms matter for humans, but not for the underlying knowledge

Conclusions

Conclusions

- We cannot solve the problem
- But we can approximate the heck out of it
- *Abstract interpretation*, *dependent types*, etc.
- Hopefully the *interesting*, *regular* programs are the only ones that we care about

Dit is het

The best of luck, and thanks for the attention!