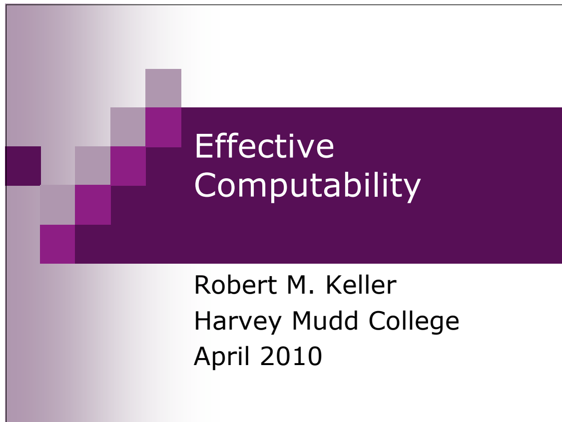
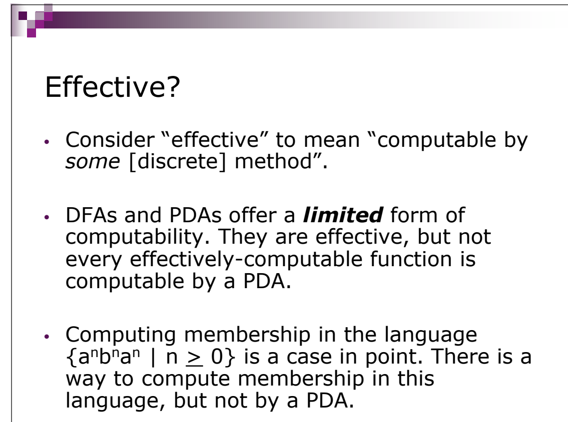
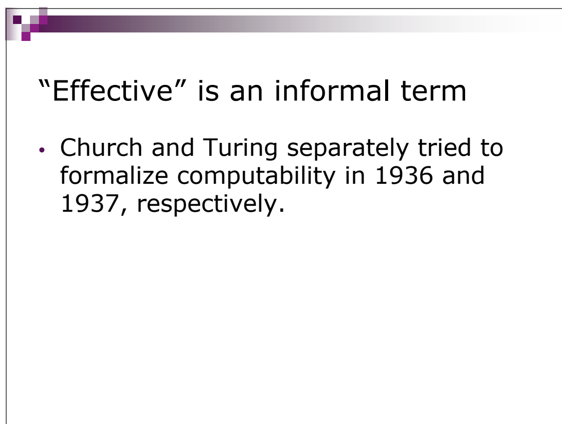
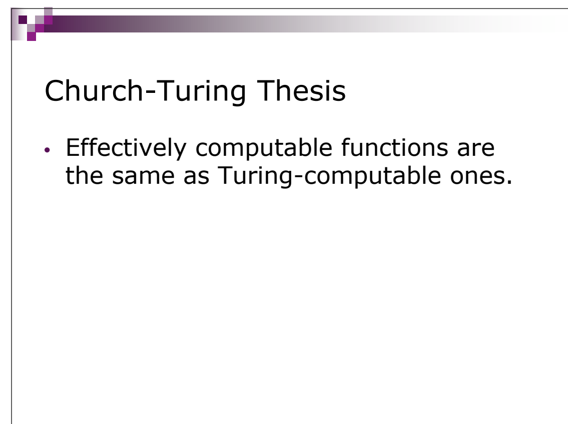
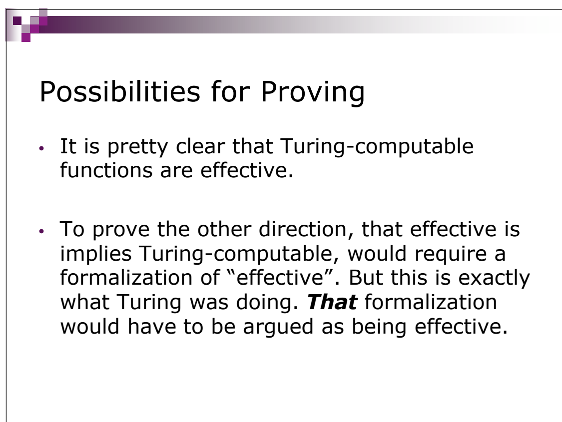
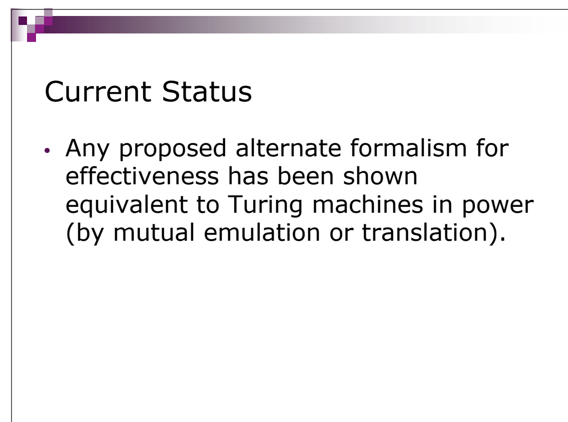

Effective Computability
Robert M. Keller
Harvey Mudd College
April 2010


Effective?
Consider "effective" to mean "computable by some [discrete] method".
DFAs and PDAs offer a limited form of computability. They are effective, but not every effectively-computable function is computable by a PDA.
Computing membership in the language {aⁿbⁿaⁿ | n ≥ 0} is a case in point. There is a way to compute membership in this language, but not by a PDA.


"Effective" is an informal term
Church and Turing separately tried to formalize computability in 1936 and 1937, respectively.


Church-Turing Thesis
Effectively computable functions are the same as Turing-computable ones.


Possibilities for Proving
It is pretty clear that Turing-computable functions are effective.
To prove the other direction, that effective is implies Turing-computable, would require a formalization of "effective". But this is exactly what Turing was doing. That formalization would have to be argued as being effective.


Current Status
Any proposed alternate formalism for effectiveness has been shown equivalent to Turing machines in power (by mutual emulation or translation).

History leading up to Church-Turing

- Richard Dedekind (1831-1916) in 1888 proposed an axiomatic foundation for the natural numbers, whose primitive notions were **1** and the **successor** function.
- Giuseppe Peano (1858–1932) in 1889 created a simpler set of axioms (also using 1), acknowledging Dedekind.

  [Using 0 vs. 1 is a non-issue, since the axioms give an *abstract* characterization of the natural numbers.
  0 is mostly used today.]

Dedekind

Peano

Wilhelm Ackermann

- Ackermann (1896-1962), a student of David Hilbert, investigated properties of recursive functions over the natural numbers.
- He defined the family of functions now known as **primitive recursive**, and found an obviously-computable function that is *not* in the family (Ackermann's function).
- He proved the consistency of a set of axioms for arithmetic, and co-authored a mathematical logic book with Hilbert.

Ackermann's Function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

$A(4,2) = 2^{65536} - 3$, an integer having 19,729 decimal digits

Kurt Gödel

Gödel (right) with Einstein

- Gödel (1906-1978) proposed in 1931 a formalism for computable functions that extended Ackermann's primitive recursive functions.
- He called these **general recursive functions**.
- These played a role in his work on representing logical formulas arithmetically, and his **incompleteness theorem**.

Alonzo Church

- Church (1903–1995) was the first to prove, in 1936, that the predicate calculus is undecidable (essentially what we did by encoding Turing machines into Otter clauses).
- This result is known as Church's Theorem.
- The problem itself then went by the name **The Entscheidungsproblem** ("Decision problem").
- Apparently his proof used some of the work of his student, Stephen Kleene.

The Lambda Calculus

- Church invented the lambda calculus in 1936. It is a system for defining **functions** and computing with them.
- It is the basis for many functional languages, including Lisp/Scheme.
- It is the residue of a **logical system** invented by Church that was eventually shown **inconsistent**, because certain expressions were equivalent to their own negation. As computation, however, this simply expresses a form of non-terminating behavior.

Let $k = (\lambda x.\ \neg(x\ x))$, then $kk = (\lambda x.\ \neg(x\ x))k = \neg(kk)$.
The function $\neg$ is somewhat arbitrary here, since kk is further applicable.

Universality of the Lambda Calculus

- Church conjectured that the lambda calculus was equivalent to effective computability.
- However, this claim was not totally convincing, and some expressed skepticism.
- Eventually the lambda calculus was shown to be equivalent to Turing machines and other formalisms.

Stephen Kleene

- Kleene (1909-1994, the same Kleene who discovered regular expressions in 1951) was a student of Church, as was Turing.
- He wrote in 1952 the text *Introduction to Metamathematics*, which expounded on Gödel's work and also introduced the **partial recursive functions**, a more structured formalism for computability.
- [This gave rise to the saying:
  "Kleeneness is next to Gödelness."]

Alan Turing

- Turing (1912-1954) was Church's student at Princeton.
- In 1937 (a year after Church published the lambda calculus), Turing published his paper "On Computable Numbers, with an Application to the Entscheidungsproblem" (Proceedings of the London Mathematical Society 42) which defined and defended a variant of what we now call the **Turing machine**, although Turing appear to be trying to characterize computability by a *human* "computer".
- Turing's defense was more intuitive and grounded than Church's defense of the lambda calculus, in my opinion.

Gödel on Turing on Computability (from Oron Shagrir, 2004)

Quoting Gödel: "The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to exactly the same concept. **The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.**" [Gödel 1951, pp. 304–305]

This is the approach Gödel recommends in his 1934 conversation with Church, in which he **rejects Church's proposal as "thoroughly unsatisfactory."** Gödel suggests to Church that "it might be possible, in terms of an effective calculability as an undefined notion, to state a set of axioms which would embody the generally accepted properties of this notion, and to do something on that basis.:

Turing's Argument

- Everyone should read the prose parts (possibly skipping some of the notation) from the original source. (It is only a few pages long.)
- Here is an html version:
  http://www.abelard.org/turpap2/tp2-ie.asp

Emil Post

- Post (1897-1954) may have been the first to prove completeness of the propositional calculus.
- In 1936, *independently of Turing*, he developed his own type of machine similar to Turing's and conjectured that it was equivalent in power to Gödel's recursive functions.
- For this and related models, see:
  http://en.wikipedia.org/wiki/Post-Turing_machine

Hao Wang

- Hao Wang (1921-1955), a student of Quine, introduced in 1954 the **B-machine** model which could **not erase symbols**, only write them, yet was equivalent in power to the Turing machine.
- [How is this possible?]
- He was the first to use a program-like **statement sequence**.

Wang's B Machine

As defined by Wang (1954) the B-machine has at its command only 4 instructions:

(1) → : Move tape-scanning head one tape square to the right (or move tape one square left), then continue to next instruction in numerical sequence;

(2) ← : Move tape-scanning head one tape square to the left (or move tape one square right), then continue to next instruction in numerical sequence;

(3) * : In scanned tape-square print mark * then go to next instruction in numerical sequence;

(4) Cn: Conditional "transfer" (jump, branch) to instruction "n": If scanned tape-square is marked then go to instruction "n" else (if scanned square is blank) continue to next instruction in numerical sequence.

Joachim Lambek's "Infinite Abacus" (1961)

- This machine used registers holding natural numbers instead of a tape holding symbols.
- This model is used extensively in logic texts by George Boolos and Richard Jeffrey.
- A host of similar models appeared around the same time, or in the decade after.
  - Counter machines (Shepherdson and Sturgis)
  - Register machines
  - RAM (Random-Access Machine)
  - RASP (Random-Access, Stored-Program) machine
- http://en.wikipedia.org/wiki/Counter_machine
  http://en.wikipedia.org/wiki/Register_machine

Extending Turing Machines

- What features could be added to Turing machines to try to make them more powerful?
  - Multiple tracks …

Extending Turing Machines

- Why don't those features make the model more powerful (as far as computability is concerned)?
  - Multiple tracks …

Universal Turing Machines

- Each Turing machine (as determined by its set of 5-tuples or rules) computes a certain function (accepts a certain language).
- A **Universal** TM can compute anything any other TM can.
- What's the catch? The UTM has to have the original TM's program written on its tape and be able to interpret it. The symbols of the original machine are encoded into symbols of the **fixed alphabet** of the UTM.

Examples of UTMs

- The Hopcroft & Ullman machine, presented in the first edition of their textbook (1969), has 40 states and 12 tape symbols, used to simulate machines with 2 tape symbols only. (This was not the first UTM.)
- Machines with more than 2 tape symbols can be transformed into ones with only 2 by encoding the symbols into binary.
- The state table of this machine is 3 pages of a text book.
- See: http://www.rdrop.com/~half/General/UTM/UTMStateTable.html

Other UTMs

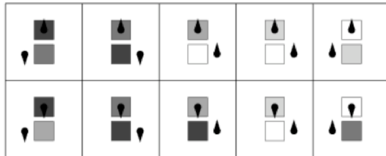
- Marvin Minsky, 1962: 4 state x 7 symbol
- (The state x symbol product is sometimes used as a measure to be reduced.)

symbols

states	1	2	3	4	5	6	7
Y	__L1	__L1	YL3	YL4	YR5	YR6	__R7
__	__L1	YR2	HALT	YR5	YL3	AL3	YR6
1	1L2	AR2	AL3	1L7	AR5	AR6	1R7
A	1L1	YR6	1L4	1L4	1R5	1R6	__R2

Other UTMs

- Stephen Wolfram, 2002: 2 state, 5 symbol machine



adapted from Wolfram, S. *A New Kind of Science*. Wolfram Media, p. 707, 2002.

TMs presented informally, Algorithms

- We typically use the Church-Turing thesis to avoid having to present a Turing machine in detail.
- If we can give a description of an **algorithm**, the thesis says that there is a TM that carries out the same computation.

Example

- There is a TM M that accepts the language $\{1^p \mid p \text{ is prime}\}$.
- Give an informal description of how M would work.

More Examples: Argue that there is an algorithm or TM for each.

- The input is an encoding of a DFA M in a straightforward way. Argue, if possible, that following problems are effectively computable:
  - For a fixed $x \in \Sigma^*$, is $x \in L(M)$?
  - For a letter $\sigma \in \Sigma$, is there an $x \in L(M)$ containing σ?
  - For a fixed $x \in \Sigma^*$, is x the prefix of some string in L(M)?
  - Is $L(M) = \Sigma^*$?
  - Is $L(M) = \varnothing$?

More Problems

- For two encoded DFAs, M and N
  - Is $L(M) \cap L(N) = \emptyset$?
  - Is $L(M) = L(N)$?
  - Is $L(M) \subseteq L(N)$?

More Problems: Is there an algorithm?

- For G a context-free grammar:
  - Is G in Chomsky normal form?
  - For a fixed $x \in \Sigma^*$, is $x \in L(G)$?
  - Is $L(G) = \emptyset$?
  - Is $L(G) = \Sigma^*$?
  - Is L(G) regular?
- For G and H two context-free grammars:
  - Is $L(G) = L(H)$?
  - Is $L(G) \cap L(H) = \emptyset$?

Decision Problems and Decidability

- The previous problems are all examples of decision problems. They are asking for a **yes/no answer**.
- All can be characterized as **language acceptance problems**.
- Example: Is L(M) = L(N)? (for DFAs): This is asking whether **a pair of encoded DFAs**, one for M and one for N, is **a member of the language** EDFA of pairs of DFA's that accept the same strings.
- **The language of interest**, EDFA, is not primarily the languages L(M) and L(N), but rather the language of encodings.

Algorithms for Decision Problems

- Consider EDFA again. Suppose a problem is represented as a pair of encodings <M><N> for M and N respectively.
- The question is whether or not <M><N> ∈ EDFA.
- Answering the question does not necessarily entail running either M nor N, but rather **analyzing** <M> and <N> for certain characteristics.