



Complexity Theory Computability

Charles E. Hughes

COT6410 – Spring 2020 Notes

Computability

The study of models of computation and what can/cannot be done via purely mechanical means

Goals of Computability

- Provide precise characterizations (computational models) of the class of effective procedures / algorithms.
- Study the boundaries between complete and incomplete models of computation.
- Study the properties of classes of solvable and unsolvable problems.
- Solve or prove unsolvable open problems.
- Determine reducibility and equivalence relations among unsolvable problems.
- Our added goal is to apply these techniques and results across multiple areas of Computer Science.

More Procedure Properties

- *Useful Notations*

- $f(x)\downarrow$ means procedure f converges/halts/produces an output, when evaluated at x .
- $f(x)\uparrow$ means procedure f diverges, when evaluated at x .
- f is an algorithm iff $\forall x f(x)\downarrow$

Sets and Decision Problems

- Set -- A collection of atoms from some universe U . \emptyset denotes the empty set.
- (Decision) Problem -- A set of questions about elements of some universe. Each question has answer “yes” or “no”. The elements having answer “yes” constitute a set that is a subset of the corresponding universe. Those having answer “no” constitute the complement of the “yes” set.

Categorizing Problems (Sets)

- Solvable or Decidable -- A problem P is said to be solvable (decidable) if there exists an algorithm F which, when applied to a question q in P , produces the correct answer (“yes” or “no”). This is an inherent property of P .
- Solved -- A problem P is said to be solved if P is solvable and we have produced its solution. This is a temporal property in that P may have been unsolved for many years before being solved.
- Unsolved, Unsolvable (Undecidable) --
Complements of above

Categorizing Problems (Sets) # 2

- Recursively enumerable -- A set S is recursively enumerable (re) if S is empty ($S = \emptyset$) or there exists an algorithm F , over the natural numbers \mathbf{N} , whose range is exactly S . A problem is said to be re if the set associated with it is re.
- Semi-Decidable -- A problem is said to be semi-decidable if there is an effective procedure F which, when applied to a question q in P , produces the answer “yes” if and only if q has answer “yes”. F need not halt if q has answer “no”.
- Semi-decidable is the same as the notion of recognizable used in the text.

Immediate Implications

- **P** solved implies **P** solvable implies **P** semi-decidable (re, recognizable).
- **P** non-re implies **P** unsolvable implies **P** unsolved.
- **P** finite implies **P** solvable.

Slightly Harder Implications

- \mathbf{P} enumerable iff \mathbf{P} semi-decidable.
- \mathbf{P} solvable iff both \mathbf{S}_P and $(\mathbf{U} - \mathbf{S}_P)$ are re (semi-decidable).

- We will prove these later.

Existence of Undecidables

- A counting argument
 - The number of mappings from N to N is at least as great as the number of subsets of N . But the number of subsets of N is uncountably infinite (\aleph_1). However, the number of programs in any model of computation is countably infinite (\aleph_0). This latter statement is a consequence of the fact that the descriptions must be finite and they must be written in a language with a finite alphabet. In fact, not only is the number of programs countable, it is also effectively enumerable; moreover, its membership is decidable.
- A diagonalization argument
 - Will be shown later in class

Hilbert's Tenth

Diophantine Equations are
Unsolvable

One Variable Diophantine
Equations are Solvable

Hilbert's 10th

- In 1900 declared there were 23 really important problems in mathematics.
- Belief was that the solutions to these would help address math's complexity.
- Hilbert's Tenth asks for an algorithm to find the integral roots of polynomials with integral coefficients. For example
 $6x^3yz^2 + 3xy^2 - x^3 - 10 = 0$ has roots
 $x = 5; y = 3; z = 0$
- This is now known to be impossible to solve (In 1970, Matiyacevič showed this undecidable).

Hilbert's 10th is Semi-Decidable

- Consider over one variable: $P(x) = 0$
- Can semi-decide by plugging in $0, 1, -1, 2, -2, 3, -3, \dots$
- This terminates and says “yes” if $P(x)$ evaluates to 0, eventually. Unfortunately, it never terminates if there is no x such that $P(x) = 0$.
- Can easily extend to $P(x_1, x_2, \dots, x_k) = 0$.

$P(x) = 0$ is Decidable

- $c_n x^n + c_{n-1} x^{n-1} + \dots + c_1 x + c_0 = 0$
- $x^n = -(c_{n-1} x^{n-1} + \dots + c_1 x + c_0)/c_n$
- $|x^n| \leq c_{\max}(|x^{n-1}| + \dots + |x| + 1)/|c_n|$
- $|x^n| \leq c_{\max}(n |x^{n-1}|)/|c_n|$, since $|x| \geq 1$
- $|x| \leq n \times c_{\max}/|c_n|$

$P(x) = 0$ is Decidable

- Can bound the search to values of x in range $[\pm n * (c_{\max} / c_n)]$, where
 n = highest order exponent in polynomial
 c_{\max} = largest absolute value coefficient
 c_n = coefficient of highest order term
- Once we have a search bound and we are dealing with a countable set, we have an algorithm to decide if there is an x .
- Cannot find bound when more than one variable, so cannot extend to $P(x_1, x_2, \dots, x_k) = 0$.

Undecidability

We Can't Do It All

Classic Unsolvable Problem

Given an arbitrary program P , in some language L , and an input x to P , will P eventually stop when run with input x ?

The above problem is called the “**Halting Problem.**” It is clearly an important and practical one – wouldn't it be nice to not be embarrassed by having your program run “forever” when you try to do a demo?

Unfortunately, there's a fly in the ointment as one can prove that no algorithm can be written in L that solves the halting problem for L .

Some terminology

We will say that a procedure, f , converges on input x if it eventually halts when it receives x as input. We denote this as $f(x)\downarrow$.

We will say that a procedure, f , diverges on input x if it never halts when it receives x as input. We denote this as $f(x)\uparrow$.

Of course, if $f(x)\downarrow$ then f defines a value for x . In fact we also say that $f(x)$ is defined if $f(x)\downarrow$ and undefined if $f(x)\uparrow$.

Finally, we define the domain of f as $\{x \mid f(x)\downarrow\}$.
The range of f is $\{y \mid f(x)\downarrow \text{ and } f(x) = y\}$.

Halting Problem

Assume we can decide the halting problem. Then there exists some total function **Halt** such that

$$\mathbf{Halt}(x,y) = \begin{cases} 1 & \text{if } \varphi_x(y) \downarrow \\ 0 & \text{if } \varphi_x(y) \uparrow \end{cases}$$

Here, we have numbered all programs and φ_x refers to the x -th program in this ordering. Now we can view Halt as a mapping from \mathbb{N} into \mathbb{N} by treating its input as a single number representing the pairing of two numbers via the one-one onto function

$$\mathbf{pair}(x,y) = \langle x,y \rangle = 2^x (2y + 1) - 1$$

with inverses

$$\langle z \rangle_1 = \log_2(z+1)$$

$$\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$$

The Contradiction

Now if **Halt** exist, then so does **Disagree**, where

$$\text{Disagree}(x) = \begin{cases} 0 & \text{if Halt}(x,x) = 0, \text{ i.e, if } \varphi_x(x) \uparrow \\ \mu y (y == y+1) & \text{if Halt}(x,x) = 1, \text{ i.e, if } \varphi_x(x) \downarrow \end{cases}$$

Since **Disagree** is a program from \aleph into \aleph , **Disagree** can be reasoned about by **Halt**. Let **d** be such that **Disagree** = φ_d , then

$$\begin{aligned} \text{Disagree}(d) \text{ is defined} & \Leftrightarrow \text{Halt}(d,d) = 0 \\ & \Leftrightarrow \varphi_d(d) \uparrow \end{aligned}$$

\Leftrightarrow **Disagree(d)** is undefined

But this means that **Disagree** contradicts its own existence. Since every step we took was constructive, except for the original assumption, we must presume that the original assumption was in error. Thus, the **Halting Problem** is not solvable.

Halting is recognizable

While the **Halting Problem** is not solvable, it is recognizable or semi-decidable.

To see this, consider the following semi-decision procedure. Let P be an arbitrary procedure and let x be an arbitrary natural number. Run the procedure P on input x until it stops. If it stops, say “yes.” If P does not stop, we will provide no answer. This semi-decides the **Halting Problem**. Here is a procedural description.

```
Semi_Decide_Halting() {  
    Read P, x;  
    P(x);  
    Print “yes”;  
}
```

Why not just algorithms?

A question that might come to mind is why we could not just have a model of computation that involves only programs that halt for all input. Assume you have such a model – our claim is that this model must be incomplete!

Here's the logic. Any programming language needs to have an associated grammar that can be used to generate all legitimate programs. By ordering the rules of the grammar in a way that generates programs in some lexical or syntactic order, we have a means to recursively enumerate the set of all programs. Thus, the set of procedures (programs) is re. using this fact, we will employ the notation that ϕ_x is the x -th procedure and $\phi_x(\mathbf{y})$ is the x -th procedure with input \mathbf{y} . We also refer to x as the procedure's index.

The universal machine

First, we can all agree that any complete model of computation must be able to simulate programs in its own language. We refer to such a simulator (interpreter) as the Universal machine, denote **Univ**. This program gets two inputs. The first is a description of the program to be simulated and the second of the input to that program. Since the set of programs in a model is re, we will assume both arguments are natural numbers; the first being the index of the program. Thus,

$$\mathbf{Univ}(x,y) = \varphi_x(y)$$

Non-re Problems

- There are even “practical” problems that are worse than unsolvable -- they’re not even semi-decidable.
- The classic non-re problem is the **Uniform Halting Problem**, that is, the problem to decide of an arbitrary effective procedure **P**, whether or not **P** is an algorithm.
- Assume that the algorithms can be enumerated, and that **F** accomplishes this. Then

$$\mathbf{F(x)} = \mathbf{F_x}$$

where $\mathbf{F_0, F_1, F_2, \dots}$ is a list of indexes of all and only the algorithms

The Contradiction

- Define $\mathbf{G}(x) = \text{Univ}(F(x), x) + 1 = \varphi_{F(x)}(x) = F_x(x) + 1$

- But then \mathbf{G} is itself an algorithm. Assume it is the \mathbf{g} -th one

$$F(\mathbf{g}) = F_{\mathbf{g}} = \mathbf{G}$$

Then,
$$\mathbf{G}(\mathbf{g}) = F_{\mathbf{g}}(\mathbf{g}) + 1 = \mathbf{G}(\mathbf{g}) + 1$$

- But then \mathbf{G} contradicts its own existence since \mathbf{G} would need to be an algorithm.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction when $\mathbf{G}(\mathbf{g})$ is undefined. In fact, we already have shown how to enumerate the (partial) recursive functions.

Consequences

- To capture all the algorithms, any model of computation must include some procedures that are not algorithms.
- Since the potential for non-termination is required, every complete model must have some form of iteration that is potentially unbounded.
- This means that simple, well-behaved for-loops (the kind where you can predict the number of iterations on entry to the loop) are not sufficient. While type loops are needed, even if implicit rather than explicit.

Insights

Non-re nature of algorithms

- No generative system (e.g., grammar) can produce descriptions of all and only algorithms
- No parsing system (even one that rejects by divergence) can accept all and only algorithms
- Of course, if you buy Church's Theorem, the set of all procedures can be generated. In fact, we can build an algorithmic acceptor of such programs.

Many unbounded ways

- How do you achieve divergence, i.e., what are the various means of unbounded computation in each of our models?
- GOTO: Turing Machines and Register Machines
- Minimization: Recursive Functions
 - Why not just simple finite iteration or recursion?
- Fixed Point: Ordered Petri Nets, (Ordered) Factor Replacement Systems

Non-determinism

- It sometimes doesn't matter
 - Turing Machines, Finite State Automata, Linear Bounded Automata
- It sometimes helps
 - Push Down Automata
- It sometimes hinders
 - Factor Replacement Systems, Petri Nets

Models of Computation

Turing Machines

Register Machines

Factor Replacement Systems

Recursive Functions

Turing Machines

1st Model

A Linear Memory Machine

Typical Textbook Description

- A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$
- Q is finite set of states
- Σ , is a finite input alphabet not containing the blank symbol \sqcup
- Γ is finite set of tape symbols that includes Σ and \sqcup commonly $\Gamma = \Sigma \cup \{\sqcup\}$
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$
- q_0 starts, q_{accept} accepts, q_{reject} rejects

Turing versus Post

- The Turing description just given requires you to write a new symbol and move off the current tape square
- Post had a variant where
$$\delta: Q \times \Gamma \rightarrow Q \times (\Gamma \cup \{R, L\})$$
- Here, you either write or move, not both
- Also, Post did not have an accept or reject state – acceptance is giving an answer of 1; rejection is 0; this treats decision procedures as predicates (functions that map input into $\{0,1\}$)
- The way we stop our machines from running is to omit actions for some discriminants making the transition function partial
- I tend to use Post's notation and to create macros so machines are easy to create
- I am not a fan of having you build Turing tables

Basic Description

- We will use a simplified form that is a variant of Post's models.
- Here, each machine is represented by a finite set of states Q , the simple alphabet $\{0,1\}$, where 0 is the blank symbol, and each state transition is defined by a 4-tuple of form

$q a X s$

where $q a$ is the discriminant based on current state q , scanned symbol a ; X can be one of $\{R, L, 0, 1\}$, signifying move right, move left, print 0, or 1; and s is the new state.

- Limiting the alphabet to $\{0,1\}$ is not really a limitation. We can represent a k -letter alphabet by encoding the j -th letter via j 1's in succession. A 0 ends each letter, and two 0's ends a word.
- We rarely write quads. Rather, we typically will build machines from simple forms.

Base Machines


- R -- move right over any scanned symbol
- L -- move left over any scanned symbol
- 0 -- write a 0 in current scanned square
- 1 -- write a 1 in current scanned square
- We can then string these machines together with optionally labeled arc.
- A labeled arc signifies a transition from one part of the composite machine to another, if the scanned square's content matches the label. Unlabeled arcs are unconditional. We will put machines together without arcs, when the arcs are unlabeled.

Useful Composite Machines

\mathcal{R} -- move right to next 0 (not including current square)

$\dots \underline{?}11\dots 10\dots \Rightarrow \dots ?11\dots 1\underline{0}\dots$ 

\mathcal{L} -- move left to next 0 (not including current square)

$\dots 011\dots 1\underline{?}\dots \Rightarrow \dots \underline{0}11\dots 1?\dots$ 

Commentary on Machines

- These machines can be used to move over encodings of letters or encodings of unary based natural numbers.
- In fact, any effective computation can easily be viewed as being over natural numbers. We can get the negative integers by pairing two natural numbers. The first is the sign (0 for +, 1 for -). The second is the magnitude.

Computing with TMs

A reasonably standard definition of a Turing computation of some n -ary function F is to assume that the machine starts with a tape containing the n inputs, x_1, \dots, x_n in the form

$$\dots 01^{x_1} 01^{x_2} 0 \dots 01^{x_n} \underline{0} \dots$$

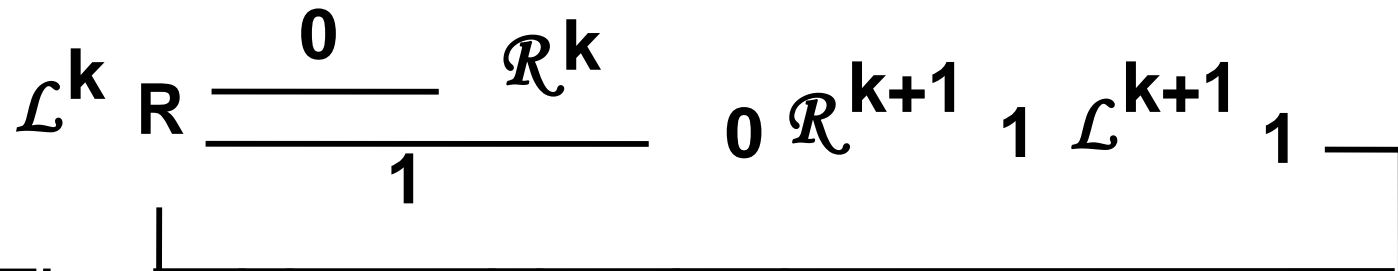
and ends with

$$\dots 01^{x_1} 01^{x_2} 0 \dots 01^{x_n} 01^y \underline{0} \dots$$

where $y = F(x_1, \dots, x_n)$.

Addition by TM

Need the copy family of useful submachines, where C_k copies k -th preceding value.



The add machine is then

$$C_2 C_2 \mathcal{L} 1 \mathcal{R} \mathcal{L} 0$$

Turing Machine Variations

- Two tracks
- N tracks
- Non-deterministic *****
- Two-dimensional
- K dimensional
- Two stack machines
- Two counter machines

Register Machines

2nd Model

Feels Like Assembly Language

Register Machine Concepts

- A register machine consists of a finite length program, each of whose instructions is chosen from a small repertoire of simple commands.
- The instructions are labeled from **1** to **m**, where there are m instructions. Termination occurs as a result of an attempt to execute the **m+1**-st instruction.
- The storage medium of a register machine is a finite set of registers, each capable of storing an arbitrary natural number.
- Any given register machine has a finite, predetermined number of registers, independent of its input.

Computing by Register Machines

- A register machine partially computing some n -ary function F typically starts with its argument values in registers 1 to n and ends with the result in the **0-th** register.
- We extend this slightly to allow the computation to start with values in its $k+1$ -st through $k+n$ -th register, with the result appearing in the k -th register, for any k , such that there are at least $k+n+1$ registers.

Register Instructions

- Each instruction of a register machine is of one of two forms:

INC_r[i] –

increment **r** and jump to **i**.

DEC_r[p, z] –

if register **r** > **0**, decrement **r** and jump to **p**

else jump to **z**

- Note, we do not use subscripts if obvious.

Addition by RM

Addition ($r0 \leftarrow r1 + r2$)

1. DEC0[1,2] : Zero result (r0) and work (r3) registers
2. DEC3[2,3]
3. DEC1[4,6] : Add r1 to r0, saving original r1 in r3
4. INC0[5]
5. INC3[3]
6. DEC3[7,8] : Restore r1
7. INC1[6]
8. DEC2[9,11] : Add r2 to r0, saving original r2 in r3
9. INC0[10]
10. INC3[8]
11. DEC3[12,13] : Restore r2
12. INC2[11]
13. : Halt by branching here

In many cases we just assume registers, other those with input, are zero at start. That would remove the need for instructions 1 and 2.

Limited Subtraction by RM

Subtraction ($r0 \leftarrow r1 - r2$, if $r1 \geq r2$; 0, otherwise)

1. DEC0[1,2] : Zero result (r0) and work (r3) registers
2. DEC3[2,3]
3. DEC1[4,6] : Add r1 to r0, saving original r1 in r3
4. INC0[5]
5. INC3[3]
6. DEC3[7,8] : Restore r1
7. INC1[6]
8. DEC2[9,11] : Subtract r2 from r0, saving original r2 in r3
9. DEC0[10,10] : Note that decrementing 0 does nothing
10. INC3[8]
11. DEC3[12,13] : Restore r2
12. INC2[11]
13. : Halt by branching here

Factor Replacement Systems

3rd Model

Deceptively Simple

Factor Replacement Concepts

- A factor replacement system (FRS) consists of a finite (ordered) sequence of fractions, and some starting natural number \mathbf{x} .
- A fraction $\mathbf{a/b}$ is applicable to some natural number \mathbf{x} , just in case \mathbf{x} is divisible by \mathbf{b} . We always chose the first applicable fraction ($\mathbf{a/b}$), multiplying it times \mathbf{x} to produce a new natural number $\mathbf{x*a/b}$. The process is then applied to this new number.
- Termination occurs when no fraction is applicable.
- A factor replacement system partially computing \mathbf{n} -ary function \mathbf{F} typically starts with its argument encoded as powers of the first \mathbf{n} odd primes. Thus, arguments $\mathbf{x_1, x_2, \dots, x_n}$ are encoded as $\mathbf{3^{x_1} 5^{x_2} \dots p_n^{x_n}}$. The result then appears as the power of the prime $\mathbf{2}$.

Addition by FRS

Addition is $3^{x1}5^{x2}$ becomes 2^{x1+x2}

or, in more details, $2^03^{x1}5^{x2}$ becomes $2^{x1+x2} 3^05^0$

$$2 / 3$$

$$2 / 5$$

Note that these systems are sometimes presented as rewriting rules of the form

$$\mathbf{bx} \rightarrow \mathbf{ax}$$

meaning that a number that has can be factored as \mathbf{bx} can have the factor \mathbf{b} replaced by an \mathbf{a} .

The previous rules would then be written

$$3x \rightarrow 2x$$

$$5x \rightarrow 2x$$

Limited Subtraction by FRS

Subtraction is $3^{x_1}5^{x_2}$ becomes $2^{\max(0, x_1-x_2)}$

$$3 \cdot 5x \rightarrow x$$

$$3x \rightarrow 2x$$

$$5x \rightarrow x$$

Ordering of Rules

- The ordering of rules are immaterial for the addition example but are critical to the workings of limited subtraction.
- In fact, if we ignore the order and just allow any applicable rule to be used, we get a form of non-determinism that makes these systems equivalent to Petri nets.
- The ordered kind are deterministic and are equivalent to a Petri net in which the transitions are prioritized.

Why Deterministic?

To see why determinism makes a difference, consider

$$3 \cdot 5x \rightarrow x$$

$$3x \rightarrow 2x$$

$$5x \rightarrow x$$

Starting with $135 = 3^3 5^1$, deterministically we get

$$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

Non-deterministically we get a larger, less selective set.

$$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

$$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$$

$$135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1$$

$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0$$

$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0$$

$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1$$

$$135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

$$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$$

...

This computes 2^z where $0 \leq z \leq x_1$. Think about it.

More on Determinism

In general, we might get an infinite set using non-determinism, whereas determinism might produce a finite set. To see this consider a system

$$2x \rightarrow x$$

$$2x \rightarrow 4x$$

starting with the number **2**.

Sample RM and FRS

Present a Register Machine that computes IsOdd. Assume $R1=x$ at starts; at termination, set $R0=1$ if x is odd; 0 otherwise. We assume $R0=0$ at start. We also are not concerned about destroying input.

1. DEC1[2, 4]
2. DEC1[1, 3]
3. INC0[4]
- 4.

Present a Factor Replacement System that computes IsOdd. Assume starting number is 3^x ; at termination, result is $2=2^1$ if x is odd; $1=2^0$ otherwise.

$$3^3 x \rightarrow x$$

$$3 x \rightarrow 2 x$$

Sample FRS

Present a Factor Replacement System that computes IsPowerOf2. Assume starting number is $3^x 5$; at termination, result is $2=2^1$ if x is a power of 2; $1=2^0$ otherwise

$$3^{2*5} x \rightarrow 5*7 x$$

$$3*5*7 x \rightarrow x$$

$$3*5 x \rightarrow 2 x$$

$$5*7 x \rightarrow 7*11 x$$

$$7*11 x \rightarrow 3*11 x$$

$$11 x \rightarrow 5 x$$

$$5 x \rightarrow x$$

$$7 x \rightarrow x$$

Systems Related to FRS

- Petri Nets:
 - Unordered
 - Ordered
 - Negated Arcs
- Vector Addition Systems:
 - Unordered
 - Ordered
- Factors with Residues:
 - $a x + c \rightarrow b x + d$
- Finitely Presented Abelian Semi-Groups

Petri Net Operation

- Finite number of places, each of which can hold zero or more markers.
- Finite number of transitions, each of which has a finite number of input and output arcs, starting and ending, respectively, at places.
- A transition is enabled if all the nodes on its input arcs have at least as many markers as arcs leading from them to this transition.
- Progress is made whenever at least one transition is enabled. Among all enabled, one is chosen randomly to fire.
- Firing a transition removes one marker per arc from the incoming nodes and adds one marker per arc to the outgoing nodes.

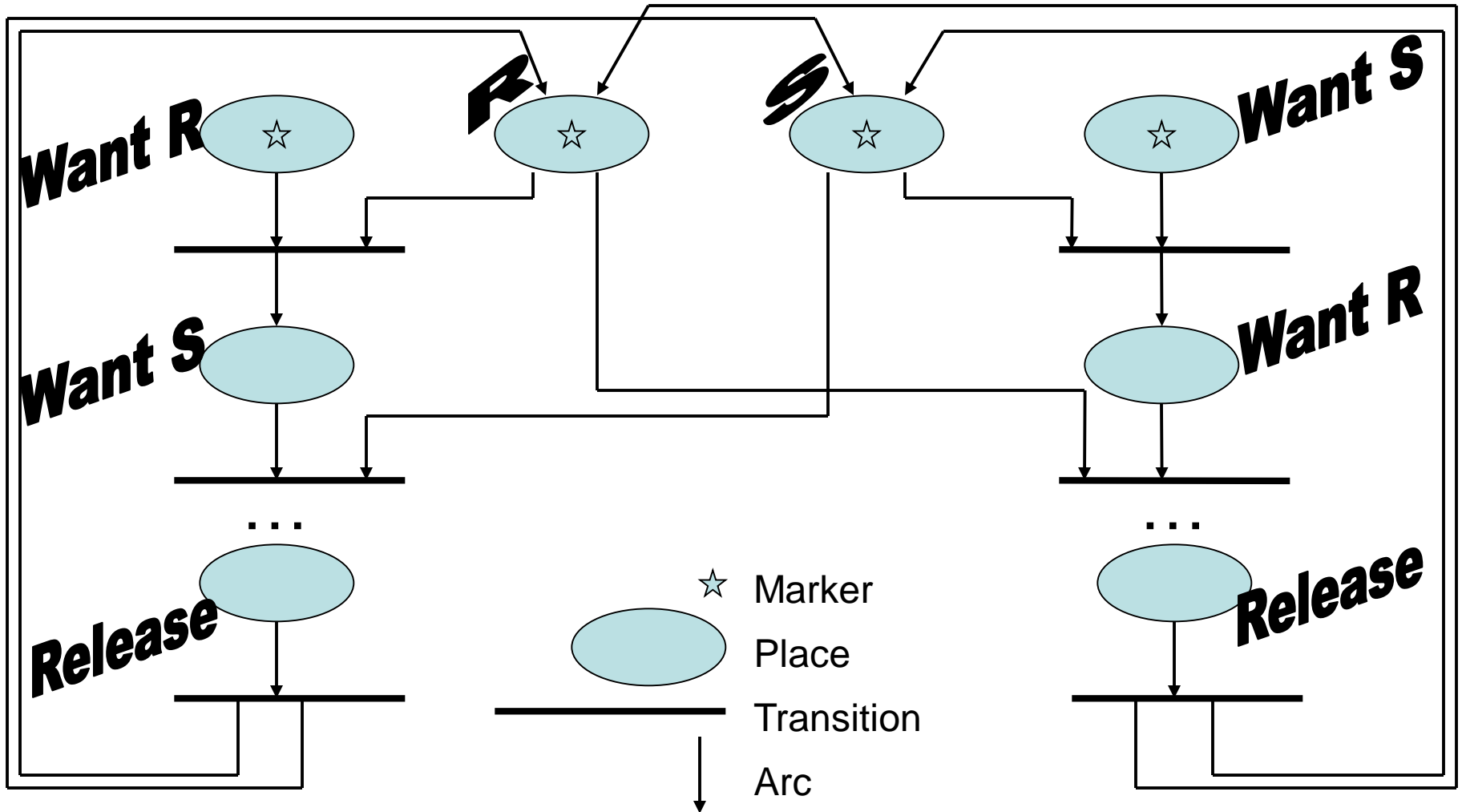
Petri Net Computation

- A Petri Net starts with some finite number of markers distributed throughout its n nodes.
- The state of the net is a vector of n natural numbers, with the i -th component's number indicating the contents of the i -th node. E.g., $\langle 0, 1, 4, 0, 6 \rangle$ could be the state of a Petri Net with **5** places, the 2nd, 3rd and 5th, having **1**, **4**, and **6** markers, resp., and the 1st and 4th being empty.
- Computation progresses by selecting and firing enabled transitions. Non-determinism is typical as many transitions can be simultaneously enabled.
- Petri nets are often used to model coordination algorithms, especially for computer networks.

Variants of Petri Nets

- A Petri Net is not computationally complete. In fact, its halting and word problems are decidable. However, its containment problem (are the markings of one net contained in those of another?) is not decidable.
- A Petri net with prioritized transitions, such that the highest priority transition is fired when multiple are enabled is equivalent to an FRS. (Think about it).
- A Petri Net with negated input arcs is one where any arc with a slash through it contributes to enabling its associated transition only if the node is empty. These are computationally complete. They can simulate register machines. (Think about this also).

Petri Net Example



Vector Addition

- Start with a finite set of vectors in integer n-space.
- Start with a single point with non-negative integral coefficients.
- Can apply a vector only if the resultant point has non-negative coefficients.
- Choose randomly among acceptable vectors.
- This generates the set of reachable points.
- Vector addition systems are equivalent to Petri Nets.
- If order vectors, these are equivalent to FRS.

Vectors as Resource Models

- Each component of a point in n -space represents the quantity of a particular resource.
- The vectors represent processes that consume and produce resources.
- The issues are safety (do we avoid bad states) and liveness (do we attain a desired state).
- Issues are deadlock, starvation, etc.

Factors with Residues

- Rules are of form
 - $a_i x + c_i \rightarrow b_i x + d_i$
 - There are n such rules
 - Can apply if number is such that you get a residue (remainder) c_i when you divide by a_i
 - Take quotient x and produce a new number $b_i x + d_i$
 - Can apply any applicable one (no order)
- These systems are equivalent to Register Machines.

Abelian Semi-Group

S = (**G**, \bullet) is a semi-group if

G is a set, \bullet is a binary operator, and

1. Closure: If $x, y \in \mathbf{G}$ then $x \bullet y \in \mathbf{G}$
2. Associativity: $x \bullet (y \bullet z) = (x \bullet y) \bullet z$

S is a monoid if

3. Identity: $\exists e \in \mathbf{G} \forall x \in \mathbf{G} [e \bullet x = x \bullet e = x]$

S is a group if

4. Inverse: $\forall x \in \mathbf{G} \exists x^{-1} \in \mathbf{G} [x^{-1} \bullet x = x \bullet x^{-1} = e]$

S is Abelian if \bullet is commutative

Finitely Presented

- $\mathbf{S} = (\mathbf{G}, \bullet)$, a semi-group (monoid, group), is finitely presented if there is a finite set of symbols, Σ , called the alphabet or generators, and a finite set of equalities $(\alpha_i = \beta_i)$, the reflexive transitive closure of which determines equivalence classes over \mathbf{G} .
- Note, the set \mathbf{G} is the closure of the generators under the semi-group's operator \bullet .
- The problem of determining membership in equivalence classes for finitely presented Abelian semi-groups is equivalent to that of determining mutual derivability in an unordered FRS or Vector Addition System with inverses for each rule.

Recursive Functions

Primitive and μ -Recursive

Primitive Recursive

An Incomplete Model

Basis of PRFs

- The primitive recursive functions are defined by starting with some base set of functions and then expanding this set via rules that create new primitive recursive functions from old ones.

- The **base functions** are:

$$\begin{aligned} \mathbf{C}_a(\mathbf{x}_1, \dots, \mathbf{x}_n) &= \mathbf{a} && : \text{constant functions} \\ \mathbf{I}_i^n(\mathbf{x}_1, \dots, \mathbf{x}_n) &= \mathbf{x}_i && : \text{identity functions} \\ &&& : \text{aka projection} \\ \mathbf{S}(\mathbf{x}) &= \mathbf{x}+1 && : \text{an increment function} \end{aligned}$$

Building New Functions

- **Composition:**

If \mathbf{G} , \mathbf{H}_1 , \dots , \mathbf{H}_k are already known to be primitive recursive, then so is \mathbf{F} , where

$$\mathbf{F}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{G}(\mathbf{H}_1(\mathbf{x}_1, \dots, \mathbf{x}_n), \dots, \mathbf{H}_k(\mathbf{x}_1, \dots, \mathbf{x}_n))$$

- **Iteration (aka primitive recursion):**

If \mathbf{G} , \mathbf{H} are already known to be primitive recursive, then so is \mathbf{F} , where

$$\mathbf{F}(\mathbf{0}, \mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{G}(\mathbf{x}_1, \dots, \mathbf{x}_n)$$

$$\mathbf{F}(\mathbf{y}+1, \mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{H}(\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{F}(\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_n))$$

We also allow definitions like the above, except iterating on \mathbf{y} as the last, rather than first argument.

Addition & Multiplication

Example: Addition

$$+(0,y) = \mathbf{I}_1^1(y)$$

$$+(x+1,y) = \mathbf{H}(x,y,+(x,y))$$

$$\text{where } \mathbf{H}(a,b,c) = \mathbf{S}(\mathbf{I}_3^3(a,b,c))$$

Example: Multiplication

$$*(0,y) = \mathbf{C}_0(y)$$

$$*(x+1,y) = \mathbf{H}(x,y,*(x,y))$$

$$\text{where } \mathbf{H}(a,b,c) = +(\mathbf{I}_2^3(a,b,c), \mathbf{I}_3^3(a,b,c))$$

$$= b+c = y + *(x,y) = (x+1)*y$$

Intuitive Composition

- Any time you have already shown some functions to be primitive recursive, you can show others are by building them up through composition
- Example#1: If g and h are primitive recursive functions (prf) then so is $f(x) = g(h(x))$. As an explicit example $\text{Add2}(x) = S(S(x)) = x+2$ is a prf
- Example#2: This can also involve multiple functions and multiple arguments like, if g , h and j are prf's then so is $f(x,y) = g(h(x), j(y))$
The problem with giving an explicit example here is that interesting compositions tend to also involve induction.

Intuitive Inductions

- A function **F** can be defined inductively using existing prf's. Typically, we have one used for the basis and another for building inductively.
- Example#1: We can build addition from successor (S)
 $x+0 = x$ (formally $+(x,0) = I(x)$)
 $x+y+1 = S(x+y)$ (more formally $+(x,y+1) = S(+(x,y))$)
- Example#2: We can build multiplication from addition
 $x*0 = 0$ (formally $*(x,0) = C_0$)
 $x*(y+1) = +(x,x*y)$ (more formally $*(x,y+1) = +(x,*(x,y))$)

Basic Arithmetic

$x + 1$:

$$x + 1 = S(x)$$

$x - 1$:

$$0 - 1 = 0$$

$$(x+1) - 1 = x$$

$x + y$:

$$x + 0 = x$$

$$x + (y+1) = (x+y) + 1$$

$x - y$: // limited subtraction

$$x - 0 = x$$

$$x - (y+1) = (x-y) - 1$$

2nd Grade Arithmetic

$x * y$:

$$\mathbf{x * 0 = 0}$$

$$\mathbf{x * (y+1) = x*y + x}$$

$x!$:

$$\mathbf{0! = 1}$$

$$\mathbf{(x+1)! = (x+1) * x!}$$

Basic Relations

$x == 0:$

$$0 == 0 = 1$$

$$(y+1) == 0 = 0$$

$x == y:$

$$x == y = ((x - y) + (y - x)) == 0$$

$x \leq y :$

$$x \leq y = (x - y) \geq 0$$

$x \geq y:$

$$x \geq y = y \leq x$$

$x > y :$

$$x > y = \sim(x \leq y) \text{ /* See } \sim \text{ on next page */}$$

$x < y :$

$$x < y = \sim(x \geq y)$$

Basic Boolean Operations

$\sim x$:

$$\sim x = 1 - x \text{ or } (x == 0)$$

signum(x): 1 if $x > 0$; 0 if $x == 0$

$$\sim(x == 0)$$

$x \&\& y$:

$$x \&\& y = \text{signum}(x * y)$$

$x \|\| y$:

$$x \|\| y = \sim((x == 0) \&\& (y == 0))$$

Definition by Cases

One case

$$f(x) = \begin{array}{ll} g(x) & \text{if } P(x) \\ h(x) & \text{otherwise} \end{array}$$

$$f(x) = P(x) * g(x) + (1-P(x)) * h(x)$$

Can use induction to prove this is true for all $k > 0$, where

$$f(x) = \begin{array}{ll} g_1(x) & \text{if } P_1(x) \\ g_2(x) & \text{if } P_2(x) \ \&\& \ \sim P_1(x) \\ \dots & \\ g_k(x) & \text{if } P_k(x) \ \&\& \ \sim(P_1(x) \ || \ \dots \ || \ \sim P_{k-1}(x)) \\ h(x) & \text{otherwise} \end{array}$$

Bounded Minimization 1

$f(x) = \mu z (z \leq x) [P(z)]$ if \exists such a z ,
= $x+1$, otherwise
where $P(z)$ is primitive recursive.

Can show f is primitive recursive by

$$\begin{aligned} f(0) &= 1 - P(0) \\ f(x+1) &= f(x) && \text{if } f(x) \leq x \\ &= x+2 - P(x+1) && \text{otherwise} \end{aligned}$$

Bounded Minimization 2

$f(x) = \mu z (z < x) [P(z)]$ if \exists such a z ,
 $= x$, otherwise
where $P(z)$ is primitive recursive.

Can show f is primitive recursive by
 $f(0) = 0$

$$f(x+1) = \mu z (z \leq x) [P(z)]$$

Intermediate Arithmetic

$x // y$:

$x // 0 = 0$: silly, but want a value

$x // (y+1) = \mu z (z < x) [(z+1) * (y+1) > x]$

$x | y$: x is a divisor of y

$x | y = ((y // x) * x) == y$

Primality

firstFactor(x): first non-zero, non-one factor of **x**.

$$\text{firstfactor}(x) = \mu z (2 \leq z \leq x) [z|x], \\ 0 \text{ if none}$$

isPrime(x):

$$\text{isPrime}(x) = \text{firstFactor}(x) == x \ \&\& \ (x > 1)$$

prime(i) = i-th prime:

$$\text{prime}(0) = 2$$

$$\text{prime}(x+1) = \mu z (\text{prime}(x) < z \leq \text{prime}(x)! + 1) [\text{isPrime}(z)]$$

We will abbreviate this as **p_i** for **prime(i)**

Exponents

x^y :

$$x^0 = 1$$

$$x^{(y+1)} = x * x^y$$

$\text{exp}(x,i)$: the exponent of p_i in number x .

$$\text{exp}(x,i) = \mu z \ (z < x) \ [\sim(p_i^{(z+1)} \mid x)]$$

Pairing Functions

- $\text{pair}(x,y) = \langle x,y \rangle = 2^x (2y + 1) - 1$

- with inverses

$$\langle z \rangle_1 = \text{exp}(z+1,0)$$

$$\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$$

- These are very useful and can be extended to encode **n**-tuples

$$\langle x,y,z \rangle = \langle x, \langle y,z \rangle \rangle \text{ (note: stack analogy)}$$

Pairing Function is 1-1 Onto

Prove that the pairing function $\langle x, y \rangle = 2^x (2y + 1) - 1$ is 1-1 onto the natural numbers.

Approach 1:

We will look at two cases, where we use the following modification of the pairing function, $\langle x, y \rangle + 1$, which implies the problem of mapping the pairing function to \mathbb{Z}^+ .

Case 1 (x=0)

Case 1:

For $x = 0$, $\langle 0, y \rangle + 1 = 2^0(2y+1) = 2y+1$. But every odd number is by definition one of the form $2y+1$, where $y \geq 0$; moreover, a particular value of y is uniquely associated with each such odd number and no odd number is produced when $x=0$. Thus, $\langle 0, y \rangle + 1$ is 1-1 onto the odd natural numbers.

Case 2 ($x > 0$)

Case 2:

For $x > 0$, $\langle x, y \rangle + 1 = 2^x(2y+1)$, where $2y+1$ ranges over all odd number and is uniquely associated with one based on the value of y (we saw that in case 1). 2^x must be even, since it has a factor of 2 and hence $2^x(2y+1)$ is also even. Moreover, from elementary number theory, we know that every even number except zero is of the form $2^x z$, where $x > 0$, z is an odd number and this pair x, y is unique. Thus, $\langle x, y \rangle + 1$ is 1-1 onto the even natural numbers, when $x > 0$.

The above shows that $\langle x, y \rangle + 1$ is 1-1 onto \mathbb{Z}^+ , but then $\langle x, y \rangle$ is 1-1 onto \mathbb{N} , as was desired.

Pairing Function is 1-1 Onto

Approach 2:

Another approach to show a function f over S is 1-1 onto T is to show that

$f^{-1}(f(x)) = x$, for arbitrary $x \in S$ and that
 $f(f^{-1}(z)) = z$, for arbitrary $z \in T$.

Thus, we need to show that

$(\langle x, y \rangle_1, \langle x, y \rangle_2) = (x, y)$ for arbitrary $(x, y) \in \mathbb{N} \times \mathbb{N}$ and
 $\langle \langle z \rangle_1, \langle z \rangle_2 \rangle = z$ for arbitrary $z \in \mathbb{N}$.

Alternate Proof

Let x, y be arbitrary natural number, then $\langle x, y \rangle = 2^x(2y+1)-1$.

Moreover, $\langle 2^x(2y+1)-1 \rangle_1 = \text{Factor}(2^x(2y+1), 0) = x$, since $2y+1$ must be odd, and

$$\langle 2^x(2y+1)-1 \rangle_2 = ((2^x(2y+1)/2^{\text{Factor}(2^x(2y+1), 0)})-1)/2 = 2y/2 = y.$$

Thus, $(\langle x, y \rangle_1, \langle x, y \rangle_2) = (x, y)$, as was desired.

Let z be an arbitrary natural number, then the inverse of the pairing is $(\langle z \rangle_1, \langle z \rangle_2)$

$$\begin{aligned} \langle \langle z \rangle_1, \langle z \rangle_2 \rangle &= 2^{\langle z \rangle_1} * (2^{\langle z \rangle_2} + 1) - 1 \\ &= 2^{\text{Factor}(z+1, 0)} * (2^{((z+1)/2^{\text{Factor}(z+1, 0)})} - 1) - 1 \\ &= 2^{\text{Factor}(z+1, 0)} * ((z+1)/2^{\text{Factor}(z+1, 0)}) - 1 \\ &= (z+1) - 1 \\ &= z, \text{ as was desired.} \end{aligned}$$

Application of Pairing

Show that prfs are closed under Fibonacci induction. Fibonacci induction means that each induction step after calculating the base is computed using the previous two values, where the previous values for $f(1)$ are $f(0)$ and 0 ; and for $x > 1$, $f(x)$ is based on $f(x-1)$ and $f(x-2)$.

The formal hypothesis is:

Assume g and h are already known to be prf, then so is f , where

$$f(0,x) = g(x);$$

$$f(1,x) = h(f(0,x), 0); \text{ and}$$

$$f(y+2,x) = h(f(y+1,x), f(y,x))$$

Proof is by construction

Fibonacci Recursion

Let K be the following primitive recursive function, defined by induction on the primitive recursive functions, g , h , and the pairing function.

$$K(0,x) = B(x)$$

$$B(x) = \langle g(x), C_0(x) \rangle \quad // \text{ this is just } \langle g(x), 0 \rangle$$

$$K(y+1, x) = J(y, x, K(y,x))$$

$$J(y,x,z) = \langle h(\langle z \rangle_1, \langle z \rangle_2), \langle z \rangle_1 \rangle$$

// this is $\langle f(y+1,x), f(y,x) \rangle$, even though f is not yet shown to be prf!!

This shows K is prf.

f is then defined from K as follows:

$$f(y,x) = \langle K(y,x) \rangle_1 \quad // \text{ extract first value from pair encoded in } K(y,x)$$

This shows it is also a prf, as was desired.

μ Recursive

4th Model

A Simple Extension to Primitive
Recursive

μ Recursive Concepts

- All primitive recursive functions are algorithms since the only iterator is bounded. That's a clear limitation.
- There are algorithms like Ackerman's function that cannot be represented by the class of primitive recursive functions.
- The class of recursive functions adds one more iterator, the minimization operator (μ), read "the least value such that."

Ackermann's Function

- $A(1, j) = 2^j$ for $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ for $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ for $i, j \geq 2$
- Wilhelm Ackermann observed in 1928 that this is not a primitive recursive function.
- Ackermann's function grows too fast to have a for-loop implementation.
- The inverse of Ackermann's function is important to analyze Union/Find algorithm. Note: $A(4,4)$ is a super exponential number involving six levels of exponentiation. $A(5,5)$ exceeds the number of atoms in known universe
- $\alpha(n) = A^{-1}(n,n)$ grows so slowly that it is less than 5 for any value of n that can be written.

Union/Find

- Start with a collection **S** of unrelated elements – singleton equivalence classes
- **Union(x,y)**, **x** and **y** are in **S**, merges the class containing **x** (**[x]**) with that containing **y** (**[y]**)
- **Find(x)** returns the canonical element of **[x]**
- Can see if **x≡y**, by seeing if **Find(x)==Find(y)**
- How do we represent the classes?

The μ Operator

- Minimization:

If \mathbf{G} is already known to be recursive, then so is \mathbf{F} , where

$$\mathbf{F}(x_1, \dots, x_n) = \mu y (\mathbf{G}(y, x_1, \dots, x_n) == 1)$$

- We also allow other predicates besides testing for one. In fact any predicate that is recursive can be used as the stopping condition.

Equivalence of Models

Equivalency of computation by
Turing machines,
register machines,
factor replacement systems,
recursive functions

Proving Equivalence

- Constructions do not, by themselves, prove equivalence.
- To do so, we need to develop a notion of an “instantaneous description” (id) of each model of computation (well, almost as recursive functions are a bit different).
- We then show a mapping of id’s between the models.

Instantaneous Descriptions

- An instantaneous description (id) is a finite description of a state achievable by a computational machine, M .
- Each machine starts in some initial id, id_0 .
- The semantics of the instructions of M define a relation \Rightarrow_M such that, $id_i \Rightarrow_M id_{i+1}$, $i \geq 0$, if the execution of a single instruction of M would alter M 's state from id_i to id_{i+1} or if M halts in state id_i and $id_{i+1} = id_i$.
- \Rightarrow^+_M is the transitive closure of \Rightarrow_M
- \Rightarrow^*_M is the reflexive transitive closure of \Rightarrow_M

id Definitions

- For a register machine, \mathbf{M} , an id is an $\mathbf{s}+1$ tuple of the form $(\mathbf{i}, r_1, \dots, r_s)_{\mathbf{M}}$ specifying the number of the next instruction to be executed and the values of all registers prior to its execution.
- For a factor replacement system, an id is just a natural number.
- For a Turing machine, \mathbf{M} , an id is some finite representation of the tape, the position of the read/write head and the current state. This is usually represented as a string $\alpha\mathbf{q}\mathbf{x}\beta$, where α (β) is the shortest string representing all non-blank squares to the left (right) of the scanned square, \mathbf{x} is the symbol at the scanned square and \mathbf{q} is the current state.
- Recursive functions do not have id's, so we will handle their simulation by an inductive argument, using the primitive functions as the basis and composition, induction and minimization in the inductive step.

Equivalence Steps

- Assume we have a machine M in one model of computation and a mapping of M into a machine M' in a second model.
- Assume the initial configuration of M is \mathbf{id}_0 and that of M' is \mathbf{id}'_0
- Define a mapping, \mathbf{h} , from id's of M into those of M' , such that, $\mathbf{R}_M = \{ \mathbf{h}(\mathbf{d}) \mid \mathbf{d} \text{ is an instance of an id of } M \}$, and
 - $\mathbf{id}'_0 \Rightarrow^*_{M'} \mathbf{h}(\mathbf{id}_0)$, and $\mathbf{h}(\mathbf{id}_0)$ is the only member of \mathbf{R}_M in the configurations encountered in this derivation.
 - $\mathbf{h}(\mathbf{id}_i) \Rightarrow^+_{M'} \mathbf{h}(\mathbf{id}_{i+1})$, $i \geq 0$, and $\mathbf{h}(\mathbf{id}_{i+1})$ is the only member of \mathbf{R}_M in this derivation.
- The above, in effect, provides an inductive proof that
 - $\mathbf{id}_0 \Rightarrow^*_M \mathbf{id}$ implies $\mathbf{id}'_0 \Rightarrow^*_{M'} \mathbf{h}(\mathbf{id})$, and
 - If $\mathbf{id}'_0 \Rightarrow^*_{M'} \mathbf{id}'$ then either $\mathbf{id}_0 \Rightarrow^*_M \mathbf{id}$, where $\mathbf{id}' = \mathbf{h}(\mathbf{id})$, or $\mathbf{id}' \notin \mathbf{R}_M$

All Models are Equivalent

Equivalency of computation by
Turing machines, register machines,
factor replacement systems,
recursive functions

Our Plan of Attack

- We will now show
**TURING \leq REGISTER \leq FACTOR \leq
RECURSIVE \leq TURING**
where, by **A \leq B**, we mean that every instance of **A** can be replaced by an equivalent instance of **B**.
- The transitive closure will then get us the desired result.

TURING \leq REGISTER

Encoding a TM's State

- Assume that we have an n state Turing machine. Let the states be numbered $0, \dots, n-1$.
- Assume our machine is in state 7 , with its tape containing
... **0 0 1 0 1 0 0 1 1 q7 0 0 0 ...**
- The underscore indicates the square being read. We denote this by the finite id
1 0 1 0 0 1 1 q7 0
- In this notation, we always write down the scanned square, even if it and all symbols to its right are blank.

More on Encoding of TM

- An id can be represented by a triple of natural numbers, (R, L, i) , where R is the number denoted by the reversal of the binary sequence to the right of the qi , L is the number denoted by the binary sequence to the left, and i is the state index.
- So,
... 0 0 1 0 1 0 0 1 1 $q7$ 0 0 0 ...
is just $(0, 83, 7)$.
... 0 0 1 0 $q5$ 1 0 1 1 0 0 ...
is represented as $(13, 2, 5)$.
- We can store the R part in register 1, the L part in register 2, and the state index in register 3.

Simulation by RM

- 1. DEC3[2,q0] : Go to simulate actions in state 0
- 2. DEC3[3,q1] : Go to simulate actions in state 1
- ...
- n. DEC3[ERR,qn-1] : Go to simulate actions in state n-1
- ...
- qj. IF_r1_ODD[qj+2] : Jump if scanning a 1
- qj+1. JUMP[set_k] : If (qj 0 0 qk) is rule in TM
- qj+1. INC1[set_k] : If (qj 0 1 qk) is rule in TM
- qj+1. DIV_r1_BY_2 : If (qj 0 R qk) is rule in TM
- MUL_r2_BY_2
- JUMP[set_k]
- qj+1. MUL_r1_BY_2 : If (qj 0 L qk) is rule in TM
- IF_r2_ODD then INC1
- DIV_r2_BY_2[set_k]
- ...
- set_n-1. INC3[set_n-2] : Set r3 to index n-1 for simulating state n-1
- set_n-2. INC3[set_n-3] : Set r3 to index n-2 for simulating state n-2
- ...
- set_0. JUMP[1] : Set r3 to index 0 for simulating state 0

Fixups

- Need epilog so action for missing quad (halting) jumps beyond end of simulation to clean things up, placing result in **r0**.
- Can also have a prolog that starts with arguments in registers **r1** to **rn** and stores values in **r1**, **r2** and **r3** to represent Turing machines starting configuration.

Prolog

Example assuming n arguments (fix as needed)

1. **MUL_{rn+1}_BY_2[2]** : Set $rn+1 = 11\dots10_2$, where, #1's = $r1$
2. **DEC1[3,4]** : $r1$ will be set to 0
3. **INC_{n+1}[1]** :
4. **MUL_{rn+1}_BY_2[5]** : Set $rn+1 = 11\dots1011\dots10_2$, where, #1's = $r1$, then $r2$
5. **DEC2[6,7]** : $r2$ will be set to 0
6. **INC_{n+1}[4]** :
- ...
- $3n-2$. **DEC_n[$3n-1, 3n+1$]** : Set $rn+1 = 11\dots1011\dots1011\dots1_2$, where, #1's = $r1, r2, \dots$
- $3n-1$. **MUL_{rn+1}_BY_2[$3n$]** : rn will be set to 0
- $3n$. **INC_{n+1}[$3n-2$]** :
- $3n+1$ **DEC_{n+1}[$3n+2, 3n+3$]** : Copy $rn+1$ to $r2$, $rn+1$ is set to 0
- $3n+2$. **INC2[$3n+1$]** :
- $3n+3$. : $r2 =$ left tape, $r1 = 0$ (right), $r3 = 0$ (initial state)

Epilog

1. **DEC3[1,2]** : Set r3 to 0 (just cleaning up)
2. **IF_r1_ODD[3,5]** : Are we done with answer?
3. **INC0[4]** : putting answer in r0
4. **DIV_r1_BY_2[2]** : strip a 1 from r1
5. : Answer is now in r0

REGISTER \leq FACTOR

Encoding a RM's State

- This is a really easy one based on the fact that every member of \mathbf{Z}^+ (the positive integers) has a unique prime factorization. Thus all such numbers can be uniquely written in the form

$$p_{i_1}^{k_1} p_{i_2}^{k_2} \cdots p_{i_j}^{k_j}$$

where the p_i 's are distinct primes and the k_i 's are non-zero values, except that the number **1** would be represented by 2^0 .

- Let R be an arbitrary **n+1**-register machine, having m instructions.

Encode the contents of registers **r0, ..., rn** by the powers of p_0, \dots, p_n .

Encode rule number's **1, ..., m** by primes p_{n+1}, \dots, p_{n+m}

Use p_{n+m+1} as prime factor that indicates simulation is done.

- This is, in essence, a Gödel number of the RM's state.

Simulation by FRS

- Now, the j -th instruction ($1 \leq j \leq m$) of R has associated factor replacement rules as follows:

j. INCr[i]

$$p_{n+j}x \rightarrow p_{n+i}p_r x$$

j. DECr[s, f]

$$p_{n+j}p_r x \rightarrow p_{n+s}x$$

$$p_{n+j}x \rightarrow p_{n+f}x$$

- We also add the halting rule associated with **m+1** of

$$p_{n+m+1}x \rightarrow x$$

Importance of Order

- The relative order of the two rules to simulate a **DEC** are critical.
- To test if register r has a zero in it, we, in effect, make sure that we cannot execute the rule that is enabled when the r -th prime is a factor.
- If the rules were placed in the wrong order, or if they weren't prioritized, we would be non-deterministic.

Sample RM and FRS (repeat)

Present a Register Machine that computes IsOdd. Assume $R1=x$ at starts; at termination, set $R0=1$ if x is odd; 0 otherwise. We assume $R0=0$ at start. We also are not concerned about destroying input.

1. DEC1[2, 4]
2. DEC1[1, 3]
3. INC0[4]
- 4.

Present a Factor Replacement System that computes IsOdd. Assume starting number is 3^x ; at termination, result is $2=2^1$ if x is odd; $1=2^0$ otherwise.

$$3^3 x \rightarrow x$$

$$3 x \rightarrow 2 x$$

Example of Order

Consider the simple machine to compute $r0 := r1 - r2$ (limited)

1. **DEC2[2,3]**
2. **DEC1[1,1]**
3. **DEC1[4,5]**
4. **INC0[3]**
- 5.

Subtraction Encoding

Start with $3^x 5^y 7^z$

$$7 \cdot 5^x \rightarrow 11^x$$

$$7^x \rightarrow 13^x$$

$$11 \cdot 3^x \rightarrow 7^x$$

$$11^x \rightarrow 7^x$$

$$13 \cdot 3^x \rightarrow 17^x$$

$$13^x \rightarrow 19^x$$

$$17^x \rightarrow 13 \cdot 2^x$$

$$19^x \rightarrow x$$

Analysis of Problem

- If we don't obey the ordering here, we could take an input like **3^55^27** and immediately apply the second rule (the one that mimics a failed decrement).
- We then have **3^55^213** , signifying that we will mimic instruction number **3**, never having subtracted the **2** from **5**.
- Now, we mimic copying **r1** to **r0** and get **2^55^219** .
- We then remove the **19** and have the wrong answer.

FACTOR \leq RECURSIVE

Universal Machine

- In the process of doing this reduction, we will build a Universal Machine.
- This is a single recursive function with two arguments. The first specifies the factor system (encoded) and the second the argument to this factor system.
- The Universal Machine will then simulate the given machine on the selected input.

Encoding FRS

- Let $(n, ((a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)))$ be some factor replacement system, where (a_i, b_i) means that the i -th rule is

$$a_i x \rightarrow b_i x$$

- Encode this machine by the number F ,

$$2^n 3^{a_1} 5^{b_1} 7^{a_2} 11^{b_2} \dots p_{2n-1}^{a_n} p_{2n}^{b_n} p_{2n+1} p_{2n+2}$$

Simulation by Recursive # 1

- We can determine the rule of **F** that applies to **x** by

$$\text{RULE}(\mathbf{F}, \mathbf{x}) = \mu z (1 \leq z \leq \text{exp}(\mathbf{F}, 0)+1) [\text{exp}(\mathbf{F}, 2*z-1) \mid \mathbf{x}]$$

- Note: $\text{exp}(\mathbf{F}, 2*i-1) = \mathbf{a}_i$ where \mathbf{a}_i is the exponent of the prime factor \mathbf{p}_{2i-1} of **F**.
- If **x** is divisible by \mathbf{a}_i , and **i** is the least integer, $1 \leq i \leq n$, for which this is true, then $\text{RULE}(\mathbf{F}, \mathbf{x}) = i$.

If **x** is not divisible by any \mathbf{a}_i , $1 \leq i \leq n$, then **x** is divisible by **1**, and $\text{RULE}(\mathbf{F}, \mathbf{x})$ returns **n+1**. That's why we added $\mathbf{p}_{2n+1} \mathbf{p}_{2n+2}$.

- Given the function $\text{RULE}(\mathbf{F}, \mathbf{x})$, we can determine $\text{NEXT}(\mathbf{F}, \mathbf{x})$, the number that follows **x**, when using **F**, by

$$\text{NEXT}(\mathbf{F}, \mathbf{x}) = (\mathbf{x} // \text{exp}(\mathbf{F}, 2*\text{RULE}(\mathbf{F}, \mathbf{x})-1)) * \text{exp}(\mathbf{F}, 2*\text{RULE}(\mathbf{F}, \mathbf{x}))$$

Simulation by Recursive # 2

- The configurations listed by F , when started on x , are
 $\text{CONFIG}(F, x, 0) = x$
 $\text{CONFIG}(F, x, y+1) = \text{NEXT}(F, \text{CONFIG}(F, x, y))$
- The number of the configuration on which F halts is
 $\text{HALT}(F, x) = \mu y [\text{CONFIG}(F, x, y) == \text{CONFIG}(F, x, y+1)]$
This assumes we converge to a fixed point as our means of halting. Of course, no applicable rule meets this definition as the $n+1$ -st rule divides and then multiplies the latest value by 1.

Simulation by Recursive # 3

- A Universal Machine that simulates an arbitrary Factor System, Turing Machine, Register Machine, Recursive Function can then be defined by

$$\text{Univ}(\mathbf{F}, x) = \text{exp} (\text{CONFIG} (\mathbf{F}, x, \text{HALT} (\mathbf{F}, x)), 0)$$

- This assumes that the answer will be returned as the exponent of the only even prime, **2**. We can fix **F** for any given Factor System that we wish to simulate. It is that ability that makes this function universal.

FRS Subtraction

- $2^0 3^a 5^b \Rightarrow 2^{a-b}$
 $3 * 5^x \rightarrow x \text{ or } 1/15$
 $5^x \rightarrow x \text{ or } 1/5$
 $3^x \rightarrow 2^x \text{ or } 2/3$
- Encode $F = 2^3 3^{15} 5^1 7^5 11^1 13^3 17^2 19^1 23^1$
- Consider $a=4, b=2$
- $RULE(F, x) = \mu z (1 \leq z \leq 4) [\exp(F, 2^{*z-1}) | x]$
 $RULE(F, 3^4 5^2) = 1$, as 15 divides $3^4 5^2$
- $NEXT(F, x) = (x // \exp(F, 2^{*RULE(F, x)-1})) * \exp(F, 2^{*RULE(F, x)})$
 $NEXT(F, 3^4 5^2) = (3^4 5^2 // 15 * 1) = 3^3 5^1$
 $NEXT(F, 3^3 5^1) = (3^3 5^1 // 15 * 1) = 3^2$
 $NEXT(F, 3^2) = (3^2 // 3 * 2) = 2^1 3^1$
 $NEXT(F, 2^1 3^1) = (2^1 3^1 // 3 * 2) = 2^2$
 $NEXT(F, 2^2) = (2^2 // 1 * 1) = 2^2$

Rest of simulation

- $\text{CONFIG}(F, x, 0) = x$
 $\text{CONFIG}(F, x, y+1) = \text{NEXT}(F, \text{CONFIG}(F, x, y))$
- $\text{CONFIG}(F, 3^4 5^2, 0) = 3^4 5^2$
 $\text{CONFIG}(F, 3^4 5^2, 1) = 3^3 5^1$
 $\text{CONFIG}(F, 3^4 5^2, 2) = 3^2$
 $\text{CONFIG}(F, 3^4 5^2, 3) = 2^1 3^1$
 $\text{CONFIG}(F, 3^4 5^2, 4) = 2^2$
 $\text{CONFIG}(F, 3^4 5^2, 5) = 2^2$
- $\text{HALT}(F, x) = \mu y [\text{CONFIG}(F, x, y) = \text{CONFIG}(F, x, y+1)] = 4$
- $\text{Univ}(F, x) = \exp(\text{CONFIG}(F, x, \text{HALT}(F, x)), 0)$
 $= \exp(2^2, 0) = 2$

Simplicity of Universal

- A side result is that every computable (recursive) function can be expressed in the form

$$F(x) = G(\mu y H(x, y))$$

where **G** and **H** are primitive recursive.

RECURSIVE \leq TURING

Standard Turing Computation

- Our notion of standard Turing computability of some n -ary function F assumes that the machine starts with a tape containing the n inputs, x_1, \dots, x_n in the form

$\dots 01^{x_1} 01^{x_2} 0 \dots 01^{x_n} \underline{0} \dots$

and ends with

$\dots 01^{x_1} 01^{x_2} 0 \dots 01^{x_n} 01^y \underline{0} \dots$

where $y = F(x_1, \dots, x_n)$.

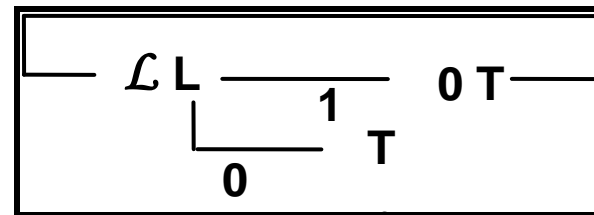
More Helpers

- To build our simulation we need to construct some useful submachines, in addition to the \mathcal{R} , \mathcal{L} , \mathbf{R} , \mathbf{L} , and \mathbf{C}_k machines already defined.

- T** -- translate moves a value left one tape square
 $\dots?01^x0\dots \Rightarrow \dots?1^x00\dots$

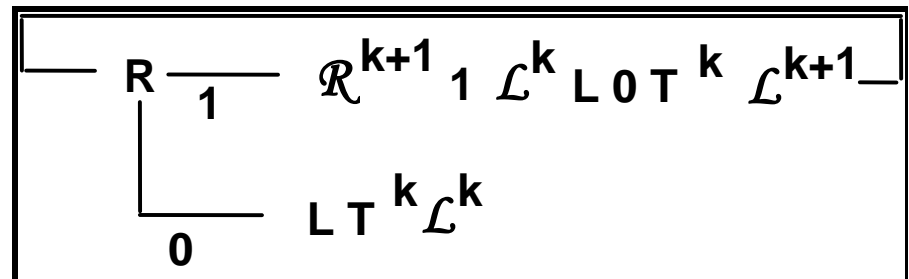


- Shift** -- shift a rightmost value left, destroying value to its left
 $\dots01^{x1}01^{x2}0\dots \Rightarrow \dots01^{x2}0\dots$



- Rot_k** -- Rotate a k value sequence one slot to the left
 $\dots01^{x1}01^{x2}0\dots01^{xk}0\dots$

$$\Rightarrow \dots01^{x2}0\dots01^{xk}01^{x1}0\dots$$



Basic Functions

All Basis Recursive Functions are Turing computable:

- $C_a^n(x_1, \dots, x_n) = a$

$$(R1)^aR$$

- $I_i^n(x_1, \dots, x_n) = x_i$

$$C_{n-i+1}$$

- $S(x) = x+1$

$$C_1 1R$$

Closure Under Composition

If $\mathbf{G}, \mathbf{H}_1, \dots, \mathbf{H}_k$ are already known to be Turing computable, then so is \mathbf{F} , where

$$\mathbf{F}(x_1, \dots, x_n) = \mathbf{G}(\mathbf{H}_1(x_1, \dots, x_n), \dots, \mathbf{H}_k(x_1, \dots, x_n))$$

To see this, we must first show that if $\mathbf{E}(x_1, \dots, x_n)$ is Turing computable then so is

$$\mathbf{E}\langle m \rangle(x_1, \dots, x_n, y_1, \dots, y_m) = \mathbf{E}(x_1, \dots, x_n)$$

This can be computed by the machine

$$\mathcal{L}^{n+m} (\text{Rot}_{n+m})^n \mathcal{R}^{n+m} \mathbf{E} \mathcal{L}^{n+m+1} (\text{Rot}_{n+m})^m \mathcal{R}^{n+m+1}$$

Can now define \mathbf{F} by

$$\mathbf{H}_1 \mathbf{H}_2 \langle 1 \rangle \mathbf{H}_3 \langle 2 \rangle \dots \mathbf{H}_k \langle k-1 \rangle \mathbf{G} \text{Shift}^k$$

Closure Under Induction

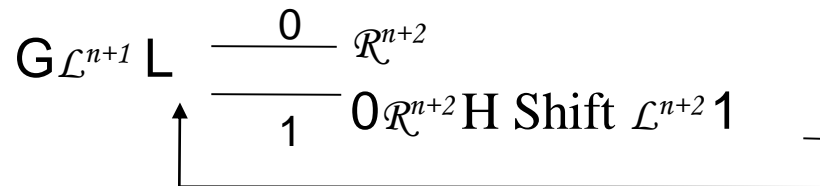
To prove that Turing Machines are closed under induction (primitive recursion), we must simulate some arbitrary primitive recursive function

$F(y, x_1, x_2, \dots, x_n)$ on a Turing Machine, where

$$F(0, x_1, x_2, \dots, x_n) = G(x_1, x_2, \dots, x_n)$$

$$F(y+1, x_1, x_2, \dots, x_n) = H(y, x_1, x_2, \dots, x_n, F(y, x_1, x_2, \dots, x_n))$$

Where, G and H are Standard Turing Computable. We define the function F for the Turing Machine as follows:



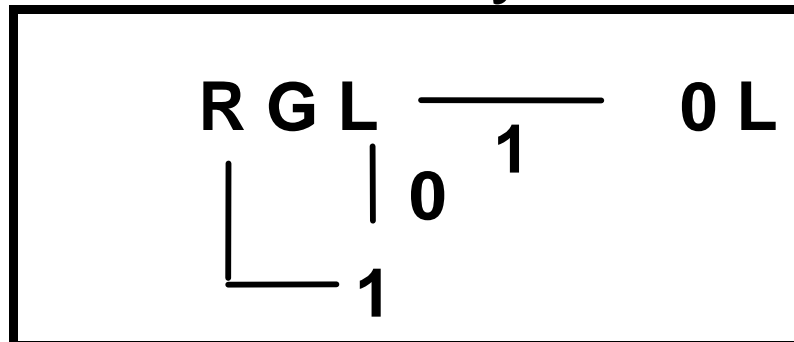
Since our Turing Machine simulator can produce the same value for any arbitrary PRF, F, we show that Turing Machines are closed under induction (primitive recursion).

Closure Under Minimization

If **G** is already known to be Turing computable, then so is **F**, where

$$\mathbf{F}(x_1, \dots, x_n) = \mu y (\mathbf{G}(x_1, \dots, x_n, y) == 1)$$

This can be done by



Consequences of Equivalence

- Theorem: The computational power of Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.
- Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.
- Theorem: Universal machines can be constructed for each of our formal models of computation.

Undecidability

We Can't Do It All

Computable Languages 1

Let's go over some important facts to this point:

1. Σ^* denotes the set of all strings over some finite alphabet Σ
2. $|\Sigma^*| = |\mathcal{N}|$, where \mathcal{N} is the set of natural numbers = the smallest infinite cardinal (the countable infinity)
3. A language L over Σ is a subset of Σ^* ; that is, $L \in \mathcal{P}(\Sigma^*) = 2^{\Sigma^*}$
Here \mathcal{P} denotes the power set constructor
4. $|L|$ is countable because $L \subseteq \Sigma^*$ (that is, $|L| \leq |\Sigma^*| = |\mathcal{N}|$)
5. $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$ (uncountable infinity) implies there are an uncountable number of languages over a given alphabet, Σ .
6. A program, P , in some programming language L , can be represented as a string over a finite alphabet, Σ_P that obeys the rules of constructing programs defined by L . As $P \in \Sigma_P^*$, there are at most a countably infinite number of programs that can be formed in the language L .

Computable Languages 2

7. Each program, P , in a programming language L , defines a function, $F_P: \Sigma_I^* \rightarrow \Sigma_O^*$ where Σ_I is the input alphabet and Σ_O is the output alphabet.
8. F_P defines an input language P_I for which F_P is defined (halts and produces an output). This is referred to as its domain in our terminology (Σ_I is its universe of discourse). The range of F_P , P_O , is the set of outputs. That is, $P_O = \{ y \mid \exists x \text{ in } P_I \text{ and } y = F_P(x) \}$
9. Since there are a countable number of programs, P , there can be at most a countable number of functions F_P and consequently, only a countable number of distinct input languages and output languages associated with programs in L_P . Thus, there are only a countable number of languages (input or output) that can be defined by any program, P .
10. But, there are an uncountable number of possible languages over any given alphabet – see 3 and 5.
11. Thus there must be languages over a given alphabet that have no descriptions – in terms of a program – or in terms of an algorithm. Thus, there are only a countably infinite number of languages that are computable among the uncountable number of possible languages.

Programming Languages

1. Programming languages that we use as software developers are in a sense “complete.” By complete we mean that they can be used to implement all procedures that we think are computable (definable by a computational model that we can “agree” covers all procedural activities).
2. **Challenge: Why did I say “agree” rather than “prove”?**
3. We mostly like programs that halt on all input (we call these algorithms), but we know it’s always possible to do otherwise in every programming language we think is complete (C, C++, C#, Java, Python, et al.)
4. We can, of course, define programming languages that define only algorithms.
5. Unfortunately, we cannot define a programming language that produces all and only algorithms (all and just programs that always halt).
6. The above (#5) is one of the main results shown in this course
7. However, before focusing on #5 we should recall that finite-state, push down and linear bounded automata are computational models that produce only algorithms (when we monitor the latter two for loops) – it’s just that these get us a subset of algorithms.

Classic Unsolvable Problem

Given an arbitrary program P , in some language L , and an input x to P , will P eventually stop when run with input x ?

The above problem is called the “Halting Problem.”

Book denotes the Halting Problem as A_{TM} .

It is clearly an important and practical one – wouldn't it be nice to not be embarrassed by having your program run “forever” when you try to do a demo for the boss or professor? Unfortunately, there's a fly in the ointment as one can prove that no algorithm can be written in L that solves the halting problem for L .

Some terminology

We will say that a procedure, f , converges on input x if it eventually halts when it receives x as input. We denote this as $f(x)\downarrow$.

We will say that a procedure, f , diverges on input x if it never halts when it receives x as input. We denote this as $f(x)\uparrow$.

Of course, if $f(x)\downarrow$ then f defines a value for x . In fact we also say that $f(x)$ is defined if $f(x)\downarrow$ and undefined if $f(x)\uparrow$.

Finally, we define the domain of f as $\{x \mid f(x)\downarrow\}$.

The range of f is $\{y \mid \text{there exists an } x, f(x)\downarrow \text{ and } f(x) = y\}$.

Numbering Procedures

Any programming language needs to have an associated grammar that can be used to generate all legitimate programs.

By ordering the rules of the grammar in a way that generates programs in some lexical or syntactic order, we have a means to recursively enumerate the set of all programs. Thus, the set of procedures (programs) is re.

Using this fact, we will employ the notation that ϕ_x is the x -th procedure and $\phi_x(\mathbf{y})$ is the x -th procedure with input \mathbf{y} . We also refer to x as the procedure's index.

The universal machine

First, we can all agree that any complete model of computation must be able to simulate programs in its own language. We refer to such a simulator (interpreter) as the Universal machine, denote Univ. This program gets two inputs. The first is a description of the program to be simulated and the second of the input to that program. Since the set of programs in a model is re, we will assume both arguments are natural numbers; the first being the index of the program. Thus,

$$\text{Univ}(x,y) = \varphi_x(y)$$

Halting Problem (A_{TM})

Assume we can decide the halting problem. Then there exists some total function Halt such that

$$\text{Halt}(x,y) = \begin{cases} 1 & \text{if } \phi_x(\mathbf{y}) \text{ is defined} \\ 0 & \text{if } \phi_x(\mathbf{y}) \text{ is not defined} \end{cases}$$

Now we can view Halt as a mapping from N into N by treating its input as a single number representing the pairing of two numbers via the one-one onto function pair discussed earlier.

$$\text{pair}(x,y) = \langle x,y \rangle = 2^x (2y + 1) - 1$$

with inverses

$$\langle z \rangle_1 = \exp(z+1, 1)$$

$$\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$$

The Contradiction

Now if Halt exist, then so does Disagree, where

$$\text{Disagree}(x) = \begin{cases} 0 & \text{if Halt}(x,x) = 0, \text{ i.e, if } \varphi_x(\mathbf{x}) \text{ is not defined} \\ \mu y (y == y+1) & \text{if Halt}(x,x) = 1, \text{ i.e, if } \varphi_x(\mathbf{x}) \text{ is defined} \end{cases}$$

Since Disagree is a program from \mathcal{N} into \mathcal{N} , Disagree can be reasoned about by Halt. Let d be such that $\text{Disagree} = [d]$, then

$$\begin{aligned} \text{Disagree}(d) \text{ is defined} & \Leftrightarrow \text{Halt}(d,d) = 0 \\ & \Leftrightarrow \varphi_d(\mathbf{d}) \text{ is undefined} \end{aligned}$$

$$\Leftrightarrow \text{Disagree}(d) \text{ is undefined}$$

But this means that Disagree contradicts its own existence. Since every step we took was constructive, except for the original assumption, we must presume that the original assumption was in error. Thus, the Halting Problem (A_{TM}) is not solvable.

Halting (A_{TM}) is recognizable

While the Halting Problem is not solvable, it is recognizable or semi-decidable.

To see this, consider the following semi-decision procedure. Let P be an arbitrary procedure and let x be an arbitrary natural number. Run the procedure P on input x until it stops. If it stops, say “yes.” If P does not stop, we will provide no answer. This semi-decides the Halting Problem. Here is a procedural description.

```
Semi_Decide_Halting() {  
    Read P, x;  
    P(x);  
    Print “yes”;  
}
```

Additional Notations

Includes comment on our notation
versus that of others

Universal Machine

- Others consider functions of n arguments, whereas we had just one. However, our input to the FRS was actually an encoding of n arguments.
- The fact that we can focus on just a single number that is the encoding of n arguments is easy to justify based on the pairing function.
- Some presentations order arguments differently, starting with the n arguments and then the Gödel number of the function, but closure under argument permutation follows from closure under substitution.

Universal Machine Mapping

- $\varphi^{(n)}(\mathbf{f}, \mathbf{x}_1, \dots, \mathbf{x}_n) = \text{Univ}(\mathbf{f}, \prod_{i=1}^n p_i^{x_i})$
- We will sometimes adopt the above and also its common shorthand

$$\varphi_{\mathbf{f}}^{(n)}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \varphi^{(n)}(\mathbf{f}, \mathbf{x}_1, \dots, \mathbf{x}_n)$$

and the even shorter version

$$\varphi_{\mathbf{f}}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \varphi^{(n)}(\mathbf{f}, \mathbf{x}_1, \dots, \mathbf{x}_n)$$

SNAP and TERM

- Our **CONFIG** is essentially a snapshot function as seen in other presentations of a universal function

$$\mathbf{SNAP(f, x, t) = CONFIG(f, x, t)}$$

- Termination in our notation occurs when we reach a fixed point, so

$$\mathbf{TERM(f, x) = (NEXT(f, x) == x)}$$

- Again, we used a single argument but that can be extended as we have already shown.

STP Predicate

- **STP**(f, x_1, \dots, x_n, t) is a predicate defined to be true iff $\phi_f(x_1, \dots, x_n)$ converges in at most t steps.
- **STP** is primitive recursive since it can be defined by
STP(f, x, t) = **TERM**($f, \text{CONFIG}(f, x, t)$)
Extending to many arguments is easily done as before.

VALUE PRF

- **VALUE(f, x1,...,xn, t)** is a primitive recursive function (algorithm) that returns $\varphi_f(x1,...,xn)$ so long as **STP(f, x1,...,xn, t)** is true.
- **VALUE(f, x1,...,xn, t) = exp (CONFIG (F, x, t), 0)**
- **VALUE(f, x1,...,xn, t)** returns a value if **STP(f, x1,...,xn, t)** is false, but the returned value is meaningless.

Recursively Enumerable

Properties of re Sets

Definition of re

- Some texts define re in the same way as I have defined semi-decidable.

$\mathbf{S} \subseteq \mathbb{N}$ is semi-decidable iff there exists a partially computable function \mathbf{g} where

$$\mathbf{S} = \{ \mathbf{x} \in \mathbb{N} \mid \mathbf{g}(\mathbf{x}) \downarrow \}$$

- I prefer the definition of re that says $\mathbf{S} \subseteq \mathbb{N}$ is re iff $\mathbf{S} = \emptyset$ or there exists a totally computable function \mathbf{f} where

$$\mathbf{S} = \{ \mathbf{y} \mid \exists \mathbf{x} \mathbf{f}(\mathbf{x}) == \mathbf{y} \}$$

- We will prove these equivalent. Actually, \mathbf{f} can be a primitive recursive function.

Semi-Decidable Implies re

Theorem: Let \mathbf{S} be semi-decided by \mathbf{G}_s . Assume \mathbf{G}_s is the g_s -th function in our enumeration of effective procedures. If $\mathbf{S} = \emptyset$ then \mathbf{S} is re by definition, so we will assume wlog that there is some $a \in \mathbf{S}$. Define the enumerating algorithm \mathbf{F}_s by

$$\mathbf{F}_s(\langle x, t \rangle) = \quad x * \mathbf{STP}(g_s, x, t) \\ \quad \quad \quad + a * (1 - \mathbf{STP}(g_s, x, t))$$

Note: \mathbf{F}_s is primitive recursive and it enumerates every value in \mathbf{S} infinitely often.

re Implies Semi-Decidable

Theorem: By definition, S is re iff $S == \emptyset$ or there exists an algorithm F_S , over the natural numbers \mathbb{N} , whose range is exactly S . Define

$\mu y [y == y+1]$ if $S == \emptyset$

$\psi_S(x) =$

$\exists y[F_S(y)==x]$, otherwise

This achieves our result as the domain of ψ_S is the range of F_S , or empty if $S == \emptyset$. Note that this is an existence proof in that we cannot test if $S == \emptyset$

Domain of a Procedure

Corollary: **S** is re/semi-decidable iff **S** is the domain / range of a partial recursive predicate F_S .

Proof: The predicate ψ_S we defined earlier to semi-decide **S**, given its enumerating function, can be easily adapted to have this property.

$$\mu y [y == y+1] \quad \text{if } S == \emptyset$$

$$\psi_S(x) =$$

$$x * \exists y [F_S(y) == x], \text{ otherwise}$$

Recursive Implies re

Theorem: Recursive implies re.

Proof: \mathbf{S} is recursive implies there is a total recursive function f_s such that

$$\mathbf{S} = \{ x \in \mathbb{N} \mid f_s(x) == 1 \}$$

Define $g_s(x) = \mu y (f_s(x) == 1)$

Clearly

$$\begin{aligned} \text{dom}(g_s) &= \{ x \in \mathbb{N} \mid g_s(x) \downarrow \} \\ &= \{ x \in \mathbb{N} \mid f_s(x) == 1 \} \\ &= \mathbf{S} \end{aligned}$$

Related Results

Theorem: \mathbf{S} is re iff \mathbf{S} is semi-decidable.

Proof: That's what we proved.

Theorem: \mathbf{S} and $\sim\mathbf{S}$ are both re (semi-decidable) iff \mathbf{S} (equivalently $\sim\mathbf{S}$) is recursive (decidable).

Proof: Let f_S semi-decide \mathbf{S} and $f_{\sim S}$ semi-decide $\sim\mathbf{S}$. We can decide \mathbf{S} by g_S

$$g_S(x) = \text{STP}(f_S, x, \mu t (\text{STP}(f_S, x, t) \parallel \text{STP}(f_{\sim S}, x, t)))$$

$$\sim\mathbf{S} \text{ is decided by } g_{\sim S}(x) = \sim g_S(x) = 1 - g_S(x).$$

The other direction is immediate since, if \mathbf{S} is decidable then $\sim\mathbf{S}$ is decidable (just complement g_S) and hence they are both re (semi-decidable).

Enumeration Theorem

- Define

$$\mathbf{W}_n = \{ \mathbf{x} \in \aleph \mid \varphi(n, \mathbf{x}) \downarrow \}$$

- Theorem: A set \mathbf{B} is re iff there exists an n such that $\mathbf{B} = \mathbf{W}_n$.

Proof: Follows from definition of $\varphi(n, \mathbf{x})$.

- This gives us a way to enumerate the recursively enumerable sets.
- Note: We will later show (again) that we cannot enumerate the recursive sets.

The Set K

- $K = \{ n \in \mathbb{N} \mid n \in W_n \}$
- Note that
 $n \in W_n \Leftrightarrow \varphi(n,n) \downarrow \Leftrightarrow \text{HALT}(n,n)$
- Thus, K is the set consisting of the indices of each program that halts when given its own index
- K can be semi-decided by the **HALT** predicate above, so it is re.

K is not Recursive

- Theorem: We can prove this by showing $\sim K$ is not re.
- If $\sim K$ is re then $\sim K = W_i$, for some i .
- However, this is a contradiction since
$$i \in K \Leftrightarrow i \in W_i \Leftrightarrow i \in \sim K \Leftrightarrow i \notin K$$

re Characterizations

Theorem: If $S \neq \emptyset$ then the following are equivalent:

1. S is re
2. S is the range of a primitive rec. function
3. S is the range of a recursive function
4. S is the range of a partial rec. function
5. S is the domain of a partial rec. function
6. S is the range/domain of a partial rec. function whose domain is the same as its range and which acts as an identity when it converges. Below, assume f_S enumerates S .

$$g_S(x) = x * \mathbf{STP}(f_S, x, \mu t (\mathbf{STP}(f_S, x, t))) \text{ or}$$

$$g_S(x) = x * \exists t \mathbf{STP}(f_S, x, t)$$

S-m-n Theorem

Parameter (S-m-n) Theorem

- Theorem: For each $n, m > 0$, there is a prf $\mathbf{S}_m^n(\mathbf{y}, \mathbf{u}_1, \dots, \mathbf{u}_n)$ such that

$$\begin{aligned} \varphi^{(m+n)}(\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_m, \mathbf{u}_1, \dots, \mathbf{u}_n) \\ = \varphi^{(m)}(\mathbf{S}_m^n(\mathbf{y}, \mathbf{u}_1, \dots, \mathbf{u}_n), \mathbf{x}_1, \dots, \mathbf{x}_m) \end{aligned}$$

- The proof of this is highly dependent on the system in which you proved universality and the encoding you chose.

S-m-n for FRS

- We would need to create a new FRS, from an existing one F , that fixes the value of u_i as the exponent of the prime p_{m+i} .
- Sketch of proof:
 Assume we normally start with $p_1^{x^1} \dots p_m^{x^m} p_1^{u^1} \dots p_{m+n}^{u^n} \sigma$
 Here the first m are variable; the next n are fixed; σ denotes prime factors used to trigger first phase of computation.
 Assume that we use fixed point as convergence.
 We start with just $p_1^{x^1} \dots p_m^{x^m}$, with q the first unused prime.

$q \alpha x \rightarrow q \beta x$	replaces $\alpha x \rightarrow \beta x$ in F , for each rule in F
$q x \rightarrow q x$	ensures we loop at end
$x \rightarrow q p_{m+1}^{u^1} \dots p_{m+n}^{u^n} \sigma x$	adds fixed input, start state and q this is selected once and never again

Note: $q = \text{prime}(\max(n+m, \text{lastFactor}(\text{Product}[i=1 \text{ to } r] \alpha_i \beta_i)) + 1)$
 where r is the number of rules in F .

Details of S-m-n for FRS

- The number of **F** (called **F**, also) is $2^r 3^{a_1} 5^{b_1} \dots p_{2r-1}^{a_r} p_{2r}^{b_r}$
- $$\mathbf{S}_{m,n}(\mathbf{F}, \mathbf{u}_1, \dots, \mathbf{u}_n) = 2^{r+2} 3^{q \times a_1} 5^{q \times b_1} \dots p_{2r-1}^{q \times a_r} p_{2r}^{q \times b_r} p_{2r+1}^q p_{2r+2}^q p_{2r+3} p_{2r+4}^q p_{m+1}^{u_1} \dots p_{m+n}^{u_n} \sigma$$
- This represents the rules we just talked about. The first added rule pair means that if the algorithm does not use fixed point, we force it to do so. The last rule pair is the only one initially enabled and it adds the prime **q**, the fixed arguments $\mathbf{u}_1, \dots, \mathbf{u}_n$, the enabling prime **q**, and the σ needed to kick start computation. Note that σ could be a **1**, if no kick start is required.
- $\mathbf{S}_{m,n} = \mathbf{S}_m^n$ is clearly primitive recursive. I'll leave the precise proof of that as a challenge to you.

Quantification 1 & 2

Quantification#1

- **S** is decidable iff there exists an algorithm χ_S (called **S**'s characteristic function) such that
$$\mathbf{x} \in \mathbf{S} \Leftrightarrow \chi_S(\mathbf{x})$$

This is just the definition of decidable.
- **S** is re iff there exists an algorithm \mathbf{A}_S where
$$\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists t \mathbf{A}_S(\mathbf{x}, t)$$

This is clear since, if g_S is the index of the procedure ψ_S that semi-decides **S** then
$$\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists t \mathbf{STP}(g_S, \mathbf{x}, t)$$

So, $\mathbf{A}_S(\mathbf{x}, t) = \mathbf{STP}_{g_S}(\mathbf{x}, t)$, where \mathbf{STP}_{g_S} is the **STP** function with its first argument fixed.
- Creating new functions by setting some one or more arguments to constants is an application of \mathbf{S}_m^n .

Quantification#2

- **S** is re iff there exists an algorithm A_S such that
 $x \notin S \Leftrightarrow \forall t A_S(x,t)$
This is clear since, if g_S is the index of the procedure ψ_S that semi-decides **S**, then
 $x \notin S \Leftrightarrow \sim \exists t \mathbf{STP}(g_S, x, t) \Leftrightarrow \forall t \sim \mathbf{STP}(g_S, x, t)$
So, $A_S(x,t) = \sim \mathbf{STP}_{g_S}(x, t)$, where \mathbf{STP}_{g_S} is the **STP** function with its first argument fixed.
- Note that this works even if **S** is recursive (decidable). The important thing there is that if **S** is recursive then it may be viewed in two normal forms, one with existential quantification and the other with universal quantification.
- The complement of an re set is co-re. A set is recursive (decidable) iff it is both re and co-re.

Diagonalization and Reducibility

Non-re Problems

- There are even “practical” problems that are worse than unsolvable -- they’re not even semi-decidable.
- The classic non-re problem is the **Uniform Halting Problem**, that is, the problem to decide of an arbitrary effective procedure **P**, whether or not **P** is an algorithm.
- Assume that the algorithms can be enumerated, and that **F** accomplishes this. Then

$$\mathbf{F(x) = F_x}$$

where $\mathbf{F_0, F_1, F_2, \dots}$ is a list of all the algorithms

The Contradiction

- Define $\mathbf{G}(x) = \text{Univ}(F(x), x) + 1 = \varphi(F(x), x) + 1 = F_x(x) + 1$
- But then \mathbf{G} is itself an algorithm. Assume it is the \mathbf{g} -th one

$$\mathbf{F}(\mathbf{g}) = F_{\mathbf{g}} = \mathbf{G}$$

Then, $\mathbf{G}(\mathbf{g}) = F_{\mathbf{g}}(\mathbf{g}) + 1 = \mathbf{G}(\mathbf{g}) + 1$

- But then \mathbf{G} contradicts its own existence since \mathbf{G} would need to be an algorithm.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction when $\mathbf{G}(\mathbf{g})$ is undefined. In fact, we already have shown how to enumerate the (partial) recursive functions.

The Set TOT

- The listing of all algorithms can be viewed as

$$\mathbf{TOT} = \{ f \in \mathbb{N} \mid \forall x \varphi(f, x) \downarrow \}$$

- We can also note that

$$\mathbf{TOT} = \{ f \in \mathbb{N} \mid W_f = \mathbb{N} \}$$

- Theorem: **TOT** is not re.

Consequences

- To capture all the algorithms, any model of computation must include some procedures that are not algorithms.
- Since the potential for non-termination is required, every complete model must have some form of iteration that is potentially unbounded.
- This means that simple, well-behaved for-loops (the kind where you can predict the number of iterations on entry to the loop) are not sufficient. While type loops are needed, even if implicit rather than explicit.

Insights

Non-re nature of algorithms

- No generative system (e.g., grammar) can produce descriptions of all and only algorithms
- No parsing system (even one that rejects by divergence) can accept all and only algorithms
- Of course, if you buy Church's Theorem, the set of all procedures can be generated. In fact, we can build an algorithmic acceptor of such programs.

Many unbounded ways

- How do you achieve divergence, i.e., what are the various means of unbounded computation in each of our models?
- GOTO: Turing Machines and Register Machines
- Minimization: Recursive Functions
 - Why not primitive recursion/iteration?
- Fixed Point: (Ordered) Factor Replacement Systems

Non-determinism

- It sometimes doesn't matter
 - Turing Machines, Finite-State Automata, Linear Bounded Automata
- It sometimes helps
 - Push Down Automata
- It sometimes hinders
 - Factor Replacement Systems, Petri Nets

Reducibility

Reduction Concepts

- Proofs by contradiction are tedious after you've seen a few. We really would like proofs that build on known unsolvable problems to show other, open problems are unsolvable. The technique commonly used is called reduction. It starts with some known unsolvable problem and then shows that this problem is no harder than some open problem in which we are interested.

Diagonalization is a Bummer

- The issues with diagonalization are that it is tedious and is applicable as a proof of undecidability or non-re-ness for only a small subset of the problems that interest us.
- Thus, we will now seek to use reduction wherever possible.
- To show a set, \mathbf{S} , is undecidable, we can show it is at least as hard as the set \mathbf{K}_0 . That is, $\mathbf{K}_0 \leq \mathbf{S}$. Here the mapping used in the reduction does not need to run in polynomial time, it just needs to be an algorithm.
- To show a set is co-re, non-recursive, we can show it is the complement of an re, non-recursive set.
- To show a set, \mathbf{S} , is not re and not even co-re, we can show it is at least as hard as the set **TOTAL (the set of algorithms)**. That is, $\mathbf{TOTAL} \leq \mathbf{S}$. We can also do this by showing it is the complement of a non-re, non-co-re set.

Reduction Example#1

- We can show that the set K_0 (Halting) is no harder than the set **TOTAL** (Uniform Halting). Since we already know that K_0 is unsolvable, we would now know that **TOTAL** is also unsolvable. We cannot reduce in the other direction since **TOTAL** is in fact harder than K_0 .
- Let φ_F be some arbitrary effective procedure and let x be some arbitrary natural number.
- Define $F_x(y) = \varphi_F(x)$, for all $y \in \mathbb{N}$
- Then F_x is an algorithm if and only if φ_F halts on x .
- Thus, $K_0 \leq \mathbf{TOTAL}$, and so a solution to membership in **TOTAL** would provide a solution to K_0 , which we know is not possible.

Reduction Examples #2 & #3

In all cases below we are assuming our variables are over \mathbb{N} .

HALT = $\{ \langle f, x \rangle \mid \varphi_f(x) \downarrow \}$ is unsolvable (undecidable, non-recursive)

TOTAL = $\{ f \mid \forall x \varphi_f(x) \downarrow \} = \{ f \mid W_f = \mathbb{N} \}$ is not even recursively enumerable (re, semidecidable)

- Show ZERO = $\{ f \mid \forall x \varphi_f(x) = 0 \}$ is unsolvable.
 $\langle f, x \rangle \in \text{HALT}$ iff $g(y) = \varphi_f(x) - \varphi_f(x)$ is zero for all y .
Thus, $\langle f, x \rangle \in \text{HALT}$ iff $g \in \text{ZERO}$ (really the index of g).
A solution to ZERO implies one for HALT, so ZERO is unsolvable.
- Show ZERO = $\{ f \mid \forall x \varphi_f(x) = 0 \}$ is non-re.
 $f \in \text{TOTAL}$ iff $h(x) = \varphi_f(x) - \varphi_f(x)$ is zero for all x .
Thus, $f \in \text{TOTAL}$ iff $h \in \text{ZERO}$ (really the index of h).
A semi-decision procedure for ZERO implies one for TOTAL, so ZERO is non-re.

Classic Undecidable Sets

- The universal language
 $K_0 = L_u = \{ \langle f, x \rangle \mid \varphi_f(x) \text{ is defined} \}$
- Membership problem for L_u is the **Halting Problem**.
- The sets L_{ne} and L_e , where

$$\text{NON-EMPTY} = L_{ne} = \{ f \mid \exists x \varphi_f(x) \downarrow \}$$

$$\text{EMPTY} = L_e = \{ f \mid \forall x \varphi_f(x) \uparrow \}$$

are the next ones we will study.

L_{ne} is re

- L_{ne} is enumerated by

$$F(\langle f, x, t \rangle) = f * \mathbf{STP}(f, x, t)$$

- This assumes that $\mathbf{0}$ is in L_{ne} since $\mathbf{0}$ probably encodes some trivial machine. If this isn't so, we'll just slightly vary our enumeration of the recursive functions so it is true.
- Thus, the range of this total function F is exactly the indices of functions that converge for some input, and that's L_{ne} .

L_{ne} is Non-Recursive

- Note in the previous enumeration that \mathbf{F} is a function of just one argument, as we are using an extended pairing function $\langle \mathbf{x}, \mathbf{y}, \mathbf{z} \rangle = \langle \mathbf{x}, \langle \mathbf{y}, \mathbf{z} \rangle \rangle$.
- Now L_{ne} cannot be recursive, for if it were then $L_u(K_0)$ is recursive by the reduction we showed before.
- In particular, from any index \mathbf{x} and input \mathbf{y} , we created a new function which accepts all input just in case the \mathbf{x} -th function accepts \mathbf{y} . Recall $\mathbf{F}_x(\mathbf{y}) = \varphi_{\mathbf{F}}(\mathbf{x})$, for all $\mathbf{y} \in \aleph$.
- Hence, this new function's index is in L_{ne} just in case $\langle \mathbf{x}, \mathbf{y} \rangle$ is in $L_u(K_0)$.
- Thus, a decision procedure for L_{ne} (equivalently for L_e) implies one for $L_u(K_0)$.

L_{ne} is re by Quantification

- Can do by observing that

$$f \in L_{ne} \Leftrightarrow \exists \langle x, t \rangle \text{ STP}(f, x, t)$$

- By our earlier results, any set whose membership can be described by an existentially quantified recursive predicate is re (semi-decidable).

L_e is not re

- If L_e were re, then L_{ne} would be recursive since it and its complement would be re.
- Can also observe that L_e is the complement of an re set since

$$\begin{aligned} f \in L_e &\Leftrightarrow \forall \langle x, t \rangle \sim \mathbf{STP}(f, x, t) \\ &\Leftrightarrow \sim \exists \langle x, t \rangle \mathbf{STP}(f, x, t) \\ &\Leftrightarrow f \notin L_{ne} \end{aligned}$$

Reduction and Equivalence

m-1, 1-1, Turing Degrees

Many-One Reduction

- Let **A** and **B** be two sets.
- We say **A** many-one reduces to **B**, $\mathbf{A} \leq_m \mathbf{B}$, if there exists a total recursive function **f** such that
$$\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{f}(\mathbf{x}) \in \mathbf{B}$$
- We say that **A** is many-one equivalent to **B**, $\mathbf{A} \equiv_m \mathbf{B}$, if $\mathbf{A} \leq_m \mathbf{B}$ and $\mathbf{B} \leq_m \mathbf{A}$
- Sets that are many-one equivalent are in some sense equally hard or easy.

Many-One Degrees

- The relationship $\mathbf{A} \equiv_m \mathbf{B}$ is an equivalence relationship (why?)
- If $\mathbf{A} \equiv_m \mathbf{B}$, we say \mathbf{A} and \mathbf{B} are of the same many-one degree (of unsolvability).
- Decidable problems occupy three m -1 degrees: \emptyset , \aleph , all others.
- The hierarchy of undecidable m -1 degrees is an infinite lattice (I'll discuss in class)

One-One Reduction

- Let **A** and **B** be two sets.
- We say **A** one-one reduces to **B**, $\mathbf{A} \leq_1 \mathbf{B}$, if there exists a total recursive 1-1 function **f** such that
$$\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{f}(\mathbf{x}) \in \mathbf{B}$$
- We say that **A** is one-one equivalent to **B**, $\mathbf{A} \equiv_1 \mathbf{B}$, if $\mathbf{A} \leq_1 \mathbf{B}$ and $\mathbf{B} \leq_1 \mathbf{A}$
- Sets that are one-one equivalent are in a strong sense equally hard or easy.

One-One Degrees

- The relationship $\mathbf{A} \equiv_1 \mathbf{B}$ is an equivalence relationship (why?)
- If $\mathbf{A} \equiv_1 \mathbf{B}$, we say \mathbf{A} and \mathbf{B} are of the same one-one degree (of unsolvability).
- Decidable problems occupy infinitely many 1-1 degrees: each cardinality defines another 1-1 degree (think about it).
- The hierarchy of undecidable 1-1 degrees is an infinite lattice.

Turing (Oracle) Reduction

- Let **A** and **B** be two sets.
- We say **A** Turing reduces to **B**, $\mathbf{A} \leq_t \mathbf{B}$, if the existence of an oracle for **B** would provide us with a decision procedure for **A**.
- We say that **A** is Turing equivalent to **B**, $\mathbf{A} \equiv_t \mathbf{B}$, if $\mathbf{A} \leq_t \mathbf{B}$ and $\mathbf{B} \leq_t \mathbf{A}$
- Sets that are Turing equivalent are in a very loose sense equally hard or easy.

Turing Degrees

- The relationship $\mathbf{A} \equiv_t \mathbf{B}$ is an equivalence relationship (why?)
- If $\mathbf{A} \equiv_t \mathbf{B}$, we say \mathbf{A} and \mathbf{B} are of the same Turing degree (of unsolvability).
- Decidable problems occupy one Turing degree. We really don't even need the oracle.
- The hierarchy of undecidable Turing degrees is an infinite lattice.

Complete re Sets

- A set **C** is re 1-1 (m-1, Turing) complete if, for any re set **A**, $A \leq_1 (\leq_m, \leq_t) C$.
- The set **HALT** is an re complete set (in regard to 1-1, m-1 and Turing reducibility).
- The re complete degree (in each sense of degree) sits at the top of the lattice of re degrees.

The Set $\text{Halt} = \mathbf{K}_0 = \mathbf{L}_u$

- $\text{Halt} = \mathbf{K}_0 = \mathbf{L}_u = \{ \langle f, x \rangle \mid \varphi_f(x) \downarrow \}$
- Let A be an arbitrary re set. By definition, there exists an effective procedure φ_a , such that $\text{dom}(\varphi_a) = A$. Put equivalently, there exists an index, a , such that $A = \mathbf{W}_a$.
- $x \in A$ iff $x \in \text{dom}(\varphi_a)$ iff $\varphi_a(x) \downarrow$ iff $\langle a, x \rangle \in \mathbf{K}_0$
- The above provides a 1-1 function that reduces A to \mathbf{K}_0 ($A \leq_1 \mathbf{K}_0$)
- Thus the universal set, $\text{Halt} = \mathbf{K}_0 = \mathbf{L}_u$, is an re (1-1, m-1, Turing) complete set.

The Set \mathbf{K}

- $\mathbf{K} = \{ f \mid \varphi_f(f) \text{ is defined} \}$
- Define $f_x(y) = \varphi_f(x)$, for all y . The index for f_x can be computed from f and x using $\mathbf{S}_{1,1}$, where we add a dummy argument, y , to φ_f . Let that index be f_x . (Yeah, that's overloading.)
- $\langle f, x \rangle \in \mathbf{K}_0$ iff $x \in \text{dom}(\varphi_f)$ iff $\forall y[\varphi_{f_x}(y) \downarrow]$ iff $f_x \in \mathbf{K}$.
- The above provides a 1-1 function that reduces \mathbf{K}_0 to \mathbf{K} .
- Since \mathbf{K}_0 is an re (1-1, m-1, Turing) complete set and \mathbf{K} is re, then \mathbf{K} is also re (1-1, m-1, Turing) complete.

Quantification # 3 and the Overall Picture

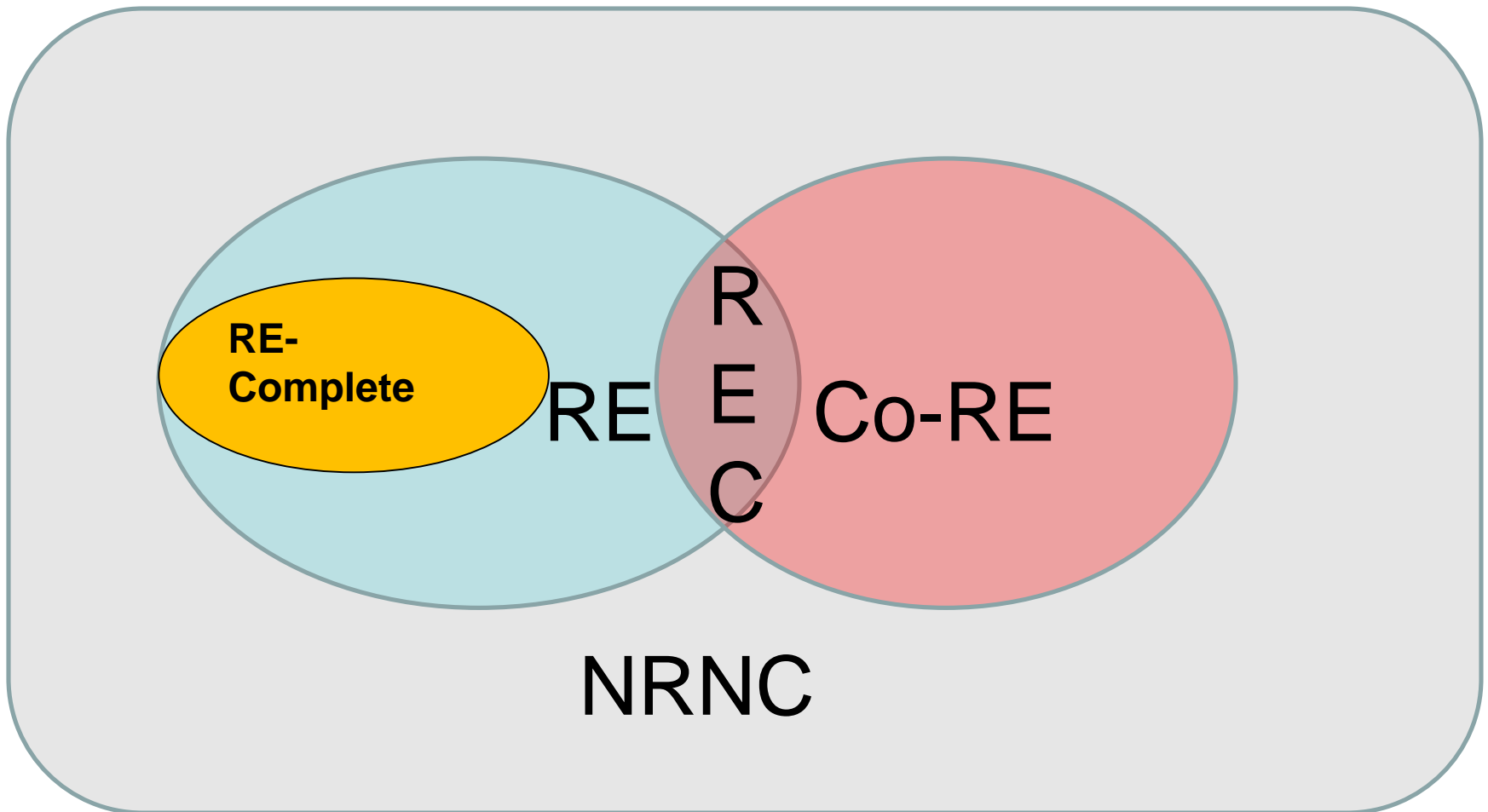
Quantification#3

- The **Uniform Halting Problem** was already shown to be non-re. It turns out its complement is also not re. We'll cover that later. In fact, we will show that **TOT** requires an alternation of quantifiers. Specifically,

$$\mathbf{f} \in \mathbf{TOT} \Leftrightarrow \forall \mathbf{x} \exists t (\mathbf{STP}(\mathbf{f}, \mathbf{x}, t))$$

and this is the minimum quantification we can use, given that the quantified predicate is total recursive (actually primitive recursive here).

UNIVERSE OF SETS



$$\text{NonRE} = (\text{NRNC} \cup \text{Co-RE}) - \text{REC}$$

Reduction and Rice's

Either Trivial or Undecidable

- Let \mathbf{P} be some set of re languages, e.g. $\mathbf{P} = \{ L \mid L \text{ is infinite re} \}$.
- We call \mathbf{P} a property of re languages since it divides the class of all re languages into two subsets, those having property \mathbf{P} and those not having property \mathbf{P} .
- \mathbf{P} is said to be trivial if it is empty (this is not the same as saying \mathbf{P} contains the empty set) or contains all re languages.
- Trivial properties are not very discriminating in the way they divide up the re languages (all or nothing).

Rice's Theorem

Rice's Theorem: Let P be some non-trivial property of the re languages. Then

$$L_P = \{ x \mid \text{dom } [x] \text{ is in } P \text{ (has property } P) \}$$

is undecidable. Note that membership in L_P is based purely on the domain of a function, not on any aspect of its implementation.

Rice's Proof-1

Proof: We will assume, *wlog*, that \mathbf{P} does not contain \emptyset . If it does we switch our attention to the complement of \mathbf{P} . Now, since \mathbf{P} is non-trivial, there exists some language \mathbf{L} with property \mathbf{P} . Let $[r]$ be a recursive function whose domain is \mathbf{L} (r is the index of a semi-decision procedure for \mathbf{L}). Suppose \mathbf{P} were decidable. We will use this decision procedure and the existence of r to decide \mathbf{K}_0 .

Rice's Proof-2

First we define a function $F_{r,x,y}$ for r and each function φ_x and input y as follows.

$$F_{r,x,y}(z) = \varphi(x, y) + \varphi(r, z)$$

The domain of this function is L if $\varphi_x(y)$ converges, otherwise it's \emptyset . Now if we can determine membership in L_P , we can use this algorithm to decide K_0 merely by applying it to $F_{r,x,y}$. An answer as to whether or not $F_{r,x,y}$ has property P is also the correct answer as to whether or not $\varphi_x(y)$ converges.

Rice's Proof-3

Thus, there can be no decision procedure for **P**.
And consequently, there can be no decision procedure for any non-trivial property of re languages.

Note: This does not apply if **P** is trivial, nor does it apply if **P** can differentiate indices that converge for precisely the same values.

I/O Property

- An I/O property, \mathcal{P} , of indices of recursive function is one that cannot differentiate indices of functions that produce precisely the same value for each input.
- This means that if two indices, \mathbf{f} and \mathbf{g} , are such that $\phi_{\mathbf{f}}$ and $\phi_{\mathbf{g}}$ converge on the same inputs and, when they converge, produce precisely the same result, then both \mathbf{f} and \mathbf{g} must have property \mathcal{P} , or neither one has this property.
- Note that any I/O property of recursive function indices also defines a property of re languages, since the domains of functions with the same I/O behavior are equal. However, not all properties of re languages are I/O properties.

Strong Rice's Theorem

Rice's Theorem: Let \mathcal{P} be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S}_{\mathcal{P}} = \{ \mathbf{x} \mid \varphi_{\mathbf{x}} \text{ has property } \mathcal{P} \}$$

is undecidable. Note that membership in $\mathbf{S}_{\mathcal{P}}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.

Strong Rice's Proof

- Given \mathbf{x} , \mathbf{y} , \mathbf{r} , where \mathbf{r} is in the set

$$\mathbf{S}_{\mathcal{P}} = \{\mathbf{f} \mid \varphi_{\mathbf{f}} \text{ has property } \mathcal{P}\},$$

define the function

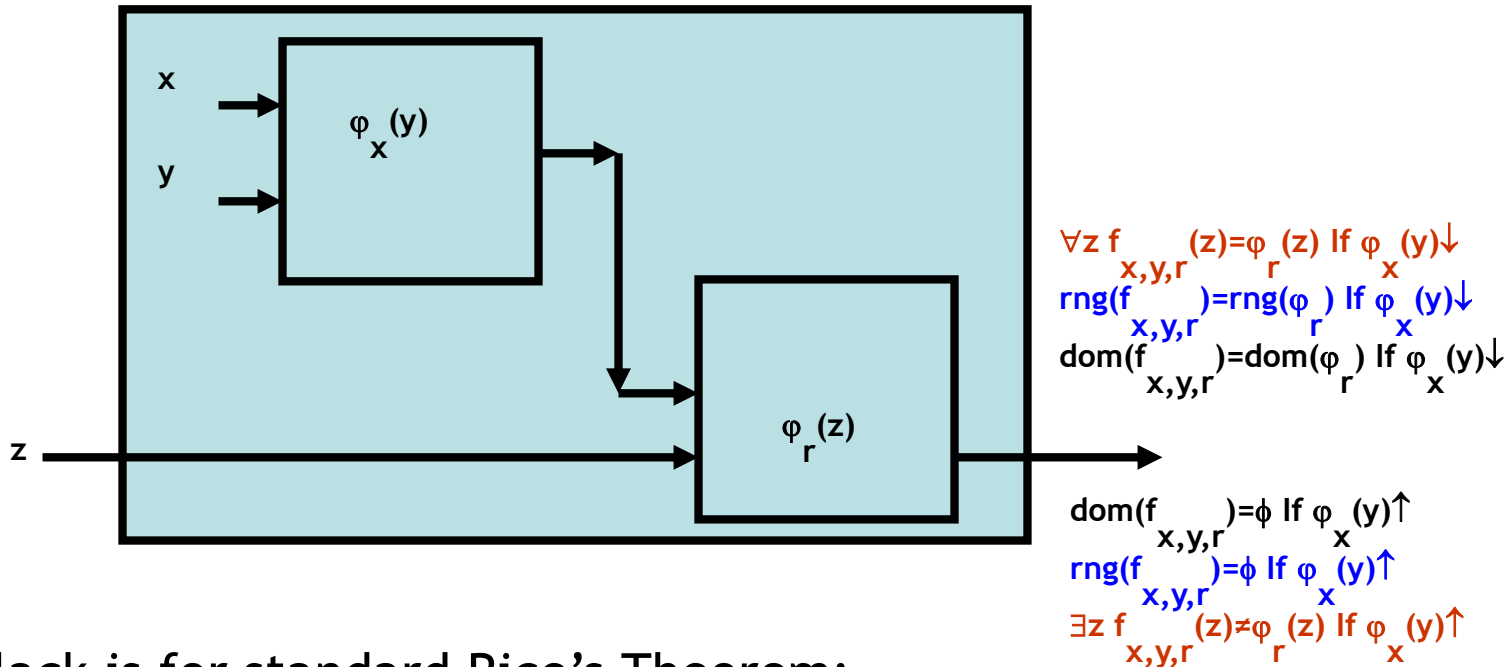
$$\mathbf{f}_{\mathbf{x},\mathbf{y},\mathbf{r}}(\mathbf{z}) = \varphi_{\mathbf{x}}(\mathbf{y}) - \varphi_{\mathbf{x}}(\mathbf{y}) + \varphi_{\mathbf{r}}(\mathbf{z}).$$

- $\mathbf{f}_{\mathbf{x},\mathbf{y},\mathbf{r}}(\mathbf{z}) = \varphi_{\mathbf{r}}(\mathbf{z})$ if $\varphi_{\mathbf{x}}(\mathbf{y}) \downarrow$; $= \phi$ if $\varphi_{\mathbf{x}}(\mathbf{y}) \uparrow$.

Thus, $\varphi_{\mathbf{x}}(\mathbf{y}) \downarrow$ iff $\mathbf{f}_{\mathbf{x},\mathbf{y},\mathbf{r}}$ has property \mathcal{P} , and so

$$\mathbf{K}_0 \leq \mathbf{S}_{\mathcal{P}}.$$

Picture Proof



Black is for standard Rice's Theorem;
 Black and Red are needed for Strong Version
 Blue is just another version based on range

Weak Rice's Theorems

Weak Rice's Theorem1: Let \mathcal{P} be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S}_{\mathcal{P}} = \{ x \mid \text{dom}(\varphi_x) \text{ has property } \mathcal{P} \}$$

is undecidable.

$$\text{dom}(f_{x,y,r}) = \text{dom}(\varphi_r) \text{ if } \varphi_x(y) \downarrow ; = \emptyset \text{ if } \varphi_x(y) \uparrow$$

Weak Rice's Theorem2: Let \mathcal{P} be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S}_{\mathcal{P}} = \{ x \mid \text{range}(\varphi_x) \text{ has property } \mathcal{P} \}$$

is undecidable.

$$\text{range}(f_{x,y,r}) = \text{range}(\varphi_r) \text{ if } \varphi_x(y) \downarrow ; = \emptyset \text{ if } \varphi_x(y) \uparrow$$

Corollaries to Rice's

Corollary: The following properties of re sets are undecidable

- a) $L = \emptyset$
- b) L is finite
- c) L is a regular set
- d) L is a context-free set

Practice

Known Results:

HALT = $\{ \langle f, x \rangle \mid f(x) \downarrow \}$ is re (semi-decidable) but undecidable

TOTAL = $\{ f \mid \forall x f(x) \downarrow \}$ is non-re (not even semi-decidable)

1. Use reduction from **HALT** to show that one cannot decide **NonTrivial**, where **NonTrivial** = $\{ f \mid \text{for some } x, y, x \neq y, f(x) \downarrow \text{ and } f(y) \downarrow \text{ and } f(x) \neq f(y) \}$
2. Show that **Non-Trivial** reduces to **HALT**. (1 plus 2 show they are equally hard)
3. Use Reduction from **TOTAL** to show that **NoRepeats** is not even re, where **NoRepeats** = $\{ f \mid \text{for all } x, y, f(x) \downarrow \text{ and } f(y) \downarrow, \text{ and } x \neq y \Rightarrow f(x) \neq f(y) \}$
4. Show **NoRepeats** reduces to **TOTAL**. (3 plus 4 show they are equally hard)
5. Use Rice's Theorem to show that **NonTrivial** is undecidable
6. Use Rice's Theorem to show that **NoRepeats** is undecidable

Practice Classifications

1. Use quantification of an algorithmic predicate to estimate the complexity (decidable, re, co-re, non-re) of each of the following, (a)-(d):
 - a) **NonTrivial** = { f | for some $x, y, x \neq y, f(x) \downarrow$ and $f(y) \downarrow$ and $f(x) \neq f(y)$ }
 - b) **NoRepeats** = { f | for all $x, y, f(x) \downarrow$ and $f(y) \downarrow$, and $x \neq y \Rightarrow f(x) \neq f(y)$ }
 - c) **FIN** = { f | domain(f) is finite }
2. Let set **A** be non-empty recursive, and let **B** be re non-recursive. Consider **C** = { z | $z = x * y$, where $x \in \mathbf{A}$ and $y \in \mathbf{B}$ }. For (a)-(c), either show sets **A** and **B** with the specified property or demonstrate that this property cannot hold.
 - a) **Can C be recursive?**
 - b) **Can C be re non-recursive (undecidable)?**
 - c) **Can C be non-re?**

Sample Question#1

1. Given that the predicate **STP** and the function **VALUE** are algorithms, show that we can semi-decide

HZ = { f | φ_f evaluates to 0 for some input }

Note: **STP(f, x, s)** is true iff $\varphi_f(\mathbf{x})$ converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = $\varphi_f(\mathbf{x})$.**

Sample Questions#2,3

2. Use Rice's Theorem to show that **HZ** is undecidable, where **HZ** is

$\text{HZ} = \{ f \mid \varphi_f \text{ evaluates to } 0 \text{ for some input} \}$

3. Redo using Reduction from **HALT**.

Sample Question#4

4. Let $\mathbf{P} = \{ f \mid \exists x [\mathbf{STP}(f, x, x)] \}$. Why does Rice's theorem not tell us anything about the undecidability of \mathbf{P} ?

Sample Question#5

5. Let **S** be an re (recursively enumerable), non-recursive set, and **T** be an re, possibly recursive non-empty set. Let

$$\mathbf{E} = \{ \mathbf{z} \mid \mathbf{z} = \mathbf{x} + \mathbf{y}, \text{ where } \mathbf{x} \in \mathbf{S} \text{ and } \mathbf{y} \in \mathbf{T} \}.$$

Answer with proofs, algorithms or counterexamples, as appropriate, each of the following questions:

- (a) Can **E** be non re?
- (b) Can **E** be re non-recursive?
- (c) Can **E** be recursive?

**Constant time:
Not amenable to Rice's**

Constant Time

- **CTime** = { **M** | $\exists K$ [**M** halts in at most **K** steps independent of its starting configuration] }
- **RT** cannot be shown undecidable by Rice's Theorem as it breaks property 2
 - Choose **M1** and **M2** to each Standard Turing Compute (STC) **ZERO**
 - **M1** is **R** (move right to end on a zero)
 - **M2** is $\mathcal{L} \mathcal{R} \mathcal{R}$ (time is dependent on argument)
 - **M1** is in **CTime**; **M2** is not, but they have same I/O behavior, so **CTime** does not adhere to property 2

Quantifier Analysis

- **CTime** = { **M** | $\exists K \forall C [\text{STP}(\mathbf{M}, \mathbf{C}, \mathbf{K})]$ }
- This would appear to imply that **CTime** is not even re. However, a TM that only runs for **K** steps can only scan at most **K** distinct tape symbols. Thus, if we use unary notation, **CTime** can be expressed
- **CTime** = { **M** | $\exists K \forall C_{|C| \leq K} [\text{STP}(\mathbf{M}, \mathbf{C}, \mathbf{K})]$ }
- We can dovetail over the set of all TMs, **M**, and all **K**, listing those **M** that halt in constant time.

Complexity of CTime

- Can show it is equivalent to the **Halting Problem** for TM's with **Infinite Tapes** (not unbounded but truly infinite)
- This was shown in 1966 to be undecidable.
- It was also shown to be re, just as we have done so for **CTime**.
- Details Later

What We've Done in Computability

List Minus Some Tedious Stuff

- A question with multiple parts that uses quantification (STP/VALUE)
- Various re and recursive equivalent definitions
- Proofs of equivalence of definitions
- Consequences of recursiveness or re-ness of a problem
- Closure of recursive/re sets
- Gödel numbering (pairing functions and inverses)
- Models of computation/equivalences (not details but understanding)
- Primitive recursion and its limitation; bounded versus unbounded μ
- Notion of universal machine
- A proof by diagonalization (there are just two possibilities)
- A question about K and/or K_0
- Many-one reduction(s)
- Rice's Theorem (its proof and its variants)
- Applications of Rice's Theorem and when it cannot be applied

More Practice Problems

Sample Question#1

1. Prove that the following are equivalent
 - a) **S is an infinite recursive (decidable) set.**
 - b) **S is the range of a monotonically increasing total recursive function.**

Note: f is monotonically increasing means that $\forall x f(x+1) > f(x)$.

Sample Question#2

2. Let A and B be re sets. For each of the following, either prove that the set is re, or give a counterexample that results in some known non-re set.

a) $A \cup B$

b) $A \cap B$

c) $\sim A$

Sample Question#3

3. Present a demonstration that the *even* function is primitive recursive.

even(x) = 1 if x is even

even(x) = 0 if x is odd

You may assume only that the base functions are prf and that prf's are closed under a finite number of applications of composition and primitive recursion.

Sample Question#4

4. Given that the predicate **STP** and the function **VALUE** are prf's, show that we can semi-decide

{ f | φ_f evaluates to 0 for some input }

Note: **STP(f, x, s)** is true iff $\varphi_f(x)$ converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = $\varphi_f(x)$** .

Sample Question#5

5. Let **S** be an re (recursively enumerable), non-recursive set, and **T** be an re, possibly recursive set. Let

$$\mathbf{E} = \{ \mathbf{z} \mid \mathbf{z} = \mathbf{x} + \mathbf{y}, \text{ where } \mathbf{x} \in \mathbf{S} \text{ and } \mathbf{y} \in \mathbf{T} \}.$$

Answer with proofs, algorithms or counterexamples, as appropriate, each of the following questions:

- (a) Can **E** be non re?
- (b) Can **E** be re non-recursive?
- (c) Can **E** be recursive?

Sample Question#6

6. Assuming that the Uniform Halting Problem (**TOTAL**) is undecidable (it's actually not even re), use reduction to show the undecidability of

$$\{ f \mid \forall x \varphi_f(x+1) > \varphi_f(x) \}$$

Sample Question#7

7. Let $\mathbf{Incr} = \{ f \mid \forall x, \varphi_f(x+1) > \varphi_f(x) \}$.
Let $\mathbf{TOT} = \{ f \mid \forall x, \varphi_f(x) \downarrow \}$.
Prove that $\mathbf{Incr} \equiv_m \mathbf{TOT}$. Note Q#6 starts this one.

Sample Question#8

8. Let $\mathbf{Incr} = \{ f \mid \forall x \varphi_f(x+1) > \varphi_f(x) \}$. Use Rice's theorem to show \mathbf{Incr} is not recursive.

Sample Question#9

9. Let \mathbf{S} be a recursive (decidable set), what can we say about the complexity (recursive, re non-recursive, non-re) of \mathbf{T} , where $\mathbf{T} \subset \mathbf{S}$?

Sample Question#10

10. Define the pairing function $\langle x, y \rangle$ and its two inverses $\langle z \rangle_1$ and $\langle z \rangle_2$, where if $z = \langle x, y \rangle$, then $x = \langle z \rangle_1$ and $y = \langle z \rangle_2$.

Sample Question#11

11. Assume $A \leq_m B$ and $B \leq_m C$.
Prove $A \leq_m C$.

Sample Question#12

12. Let $\mathbf{P} = \{ f \mid \exists x [\mathbf{STP}(f, x, x)] \}$. Why does Rice's theorem not tell us anything about the undecidability of \mathbf{P} ?

Rewriting Systems

Post Systems

Thue Systems

- Devised by Axel Thue
- Just a string rewriting view of finitely presented monoids
- $T = (\Sigma, R)$, where Σ is a finite alphabet and R is a finite set of bi-directional rules of form $\alpha_i \leftrightarrow \beta_i$, $\alpha_i, \beta_i \in \Sigma^*$
- We define \Leftrightarrow^* as the reflexive, transitive closure of \Leftrightarrow , where $w \Leftrightarrow x$ iff $w = y\alpha z$ and $x = y\beta z$, where $\alpha \leftrightarrow \beta$

Semi-Thue Systems

- Devised by Emil Post
- A one-directional version of Thue systems
- $S = (\Sigma, R)$, where Σ is a finite alphabet and R is a finite set of rules of form $\alpha_i \rightarrow \beta_i$, $\alpha_i, \beta_i \in \Sigma^*$
- We define \Rightarrow^* as the reflexive, transitive closure of \Rightarrow , where $w \Rightarrow x$ iff $w = y\alpha z$ and $x = y\beta z$, where $\alpha \rightarrow \beta$

Word Problems

- Let $S = (\Sigma, R)$ be some Thue (Semi-Thue) system, then the word problem for S is the problem to determine of arbitrary words w and x over S , whether or not $w \Leftrightarrow^* x$ ($w \Rightarrow^* x$)
- The Thue system word problem is the problem of determining membership in equivalence classes. This is not true for Semi-Thue systems.
- We can always consider just the relation \Rightarrow^* since the symmetric property of \Leftrightarrow^* comes directly from the rules of Thue systems.

Post Canonical Systems

- These are a generalization of Semi-Thue systems.
- $P = (\Sigma, V, R)$, where Σ is a finite alphabet, V is a finite set of “variables”, and R is a finite set of rules.
- Here the premise part (left side) of a rule can have many premise forms, e.g, a rule appears as

$$\begin{array}{l} \alpha_{1,0} P_{1,1} \alpha_{1,1} P_{1,2} \cdots \alpha_{1,n_1} P_{1,n_1} \alpha_{1,n_1+1} , \\ \alpha_{2,0} P_{2,1} \alpha_{2,1} P_{2,2} \cdots \alpha_{2,n_2} P_{2,n_2} \alpha_{2,n_2+1} , \\ \dots \\ \alpha_{k,0} P_{k,1} \alpha_{k,1} P_{k,2} \cdots \alpha_{k,n_k} P_{k,n_k} \alpha_{k,n_k+1} , \\ \rightarrow \beta_0 Q_1 \beta_1 Q_2 \cdots \beta_{n_{k+1}} Q_{n_{k+1}} \beta_{n_{k+1}+1} \end{array}$$
- In the above, the P 's and Q 's are variables, the α 's and β 's are strings over Σ , and each Q must appear in at least one premise.
- We can extend the notion of \Rightarrow^* to these systems considering sets of words that derive conclusions. Think of the original set as axioms, the rules as inferences and the final word as a theorem to be proved.

Examples of Canonical Forms

- Propositional rules

$$P, P \supset Q \rightarrow Q$$

$$\sim P, P \cup Q \rightarrow Q$$

$$P \cap Q \rightarrow P$$

$$P \cap Q \rightarrow Q$$

$$(P \cap Q) \cap R \leftrightarrow P \cap (Q \cap R)$$

$$(P \cup Q) \cup R \leftrightarrow P \cup (Q \cup R)$$

$$\sim(\sim P) \leftrightarrow P$$

$$P \cup Q \rightarrow Q \cup P$$

$$P \cap Q \rightarrow Q \cap P$$

oh, oh $a \cap (b \cap c) \Rightarrow a \cap (b$

- Some proofs over $\{a, b, (,), \sim, \supset, \cup, \cap\}$

$$\{a \cup c, b \supset \sim c, b\} \Rightarrow \{a \cup c, b \supset \sim c, b, \sim c\} \Rightarrow$$

$$\{a \cup c, b \supset \sim c, b, \sim c, c \cup a\} \Rightarrow$$

$$\{a \cup c, b \supset \sim c, b, \sim c, c \cup a, a\} \text{ which proves "a"}$$

Simplified Canonical Forms

- Each rule of a Semi-Thue system is a canonical rule of the form
 $P\alpha Q \rightarrow P\beta Q$
- Each rule of a Thue system is a canonical rule of the form
 $P\alpha Q \leftrightarrow P\beta Q$
- Each rule of a Post Normal system is a canonical rule of the form
 $\alpha P \rightarrow P\beta$
- Tag systems are just Normal systems where all premises are of the same length (the deletion number), and at most one can begin with any given letter in Σ . That makes Tag systems deterministic.

Examples of Post Systems

- Alphabet $\Sigma = \{a,b,\#\}$. Semi-Thue rules:
aba \rightarrow b
#b# $\rightarrow \lambda$
For above, $\#a^nba^m\# \Rightarrow^* \lambda$ iff $n=m$
- Alphabet $\Sigma = \{0,1,c,\#\}$. Normal rules:
0c \rightarrow 1
1c \rightarrow c0
#c \rightarrow #1
0 \rightarrow 0
1 \rightarrow 1
\rightarrow #
For above, $binaryc\# \Rightarrow^* binary+1\#$ where *binary* is some binary number.

Simulating Turing Machines

- Basically, we need at least one rule for each 4-tuple in the Turing machine's description.
- The rules lead from one instantaneous description to another.
- The Turing ID $\alpha q a \beta$ is represented by the string $h \alpha q a \beta h$, a being the scanned symbol.
- The tuple $q a b s$ leads to $q a \rightarrow s b$
- Moving right and left can be harder due to blanks.

Details of $\text{Halt(TM)} \leq \text{Word(ST)}$

- Let $M = (Q, \{0,1\}, T)$, T is Turing table.
- If $qabs \in T$, add rule $qa \rightarrow sb$ // simple rewrite of scan
- If $qaRs \in T$, add rules
 - $q1b \rightarrow 1sb$ $a=1, \forall b \in \{0,1\}$ // left non-blank; scan not blank
 - $q1h \rightarrow 1s0h$ $a=1$ // right blank; scan not blank
 - $cq0b \rightarrow c0sb$ $a=0, \forall b,c \in \{0,1\}$ // left and right non-blank; scan blank
 - $hq0b \rightarrow hsb$ $a=0, \forall b \in \{0,1\}$ // left blank; right not blank; scan blank
 - $cq0h \rightarrow c0s0h$ $a=0, \forall c \in \{0,1\}$ // left not blank; right blank; scan blank
 - $hq0h \rightarrow hs0h$ $a=0$ // blank tape to blank tape
- If $qaLs \in T$, add rules
 - $bqac \rightarrow sbac$ $\forall a,b,c \in \{0,1\}$ // left and right had non-blanks
 - $hqac \rightarrow hs0ac$ $\forall a,c \in \{0,1\}$ // left blank; right not blank
 - $bq1h \rightarrow sb1h$ $a=1, \forall b \in \{0,1\}$ // left not blank; right blank; scan not blank
 - $hq1h \rightarrow hs01h$ $a=1$ // left blank; right blank; scan not blank
 - $bq0h \rightarrow sbh$ $a=0, \forall b \in \{0,1\}$ // left not blank; right blank; scan blank
 - $hq0h \rightarrow hs0h$ $a=0$ // blank tape to blank tape

Clean-Up

- Assume q_1 is start state and only one accepting state exists q_0
- We will start in $h1^xq_10h$, seeking to accept x (enter q_0) or reject (run forever).
- Add rules
 - $q_0a \rightarrow q_0$ $\forall a \in \{0,1\}$
 - $bq_0 \rightarrow q_0$ $\forall b \in \{0,1\}$
- The added rule allows us to “erase” the tape if we accept x .
- This means that acceptance can be changed to generating hq_0h .
- The next slide shows the consequences.

Semi-Thue Word Problem

- Construction from TM, M , gets:
- $h1^xq_10h \Rightarrow_{\Sigma(M)^*} hq_0h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Rightarrow_{\Pi(M)^*} h1^xq_10h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Leftrightarrow_{\Sigma(M)^*} h1^xq_10h$ iff $x \in \mathcal{L}(M)$.
- Can recast both Semi-Thue and Thue Systems to ones over alphabet $\{a,b\}$ or $\{0,1\}$. That is, a binary alphabet is sufficient for undecidability.

More on Grammars

Grammars and re Sets

- Every grammar lists an re set.
- Some grammars (regular, CFL and CSG) produce recursive sets.
- Type 0 grammars are as powerful at generating (producing) re sets as Turing machines are at enumerating them (Proof later).

Formal Language

Undecidability Continued

PCP and Traces

Post Correspondence Problem

- Many problems related to grammars can be shown to be no more complex than the Post Correspondence Problem (PCP).
- Each instance of PCP is denoted: Given $n > 0$, S a finite alphabet, and two n -tuples of words $(x_1, \dots, x_n), (y_1, \dots, y_n)$ over S , does there exist a sequence $i_1, \dots, i_k, k > 0, 1 \leq i_j \leq n$, such that
$$x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k} ?$$
- Example of PCP:
 $n = 3, S = \{a, b\}, (aba, bb, a), (bab, b, baa)$.
Solution 2, 3, 1, 2
 $bb \ a \ a \ ba \ bb = b \ ba \ a \ bab \ b$

PCP Example#2

- Start with Semi-Thue System

- $aba \rightarrow ab; a \rightarrow aa; b \rightarrow a$

- Instance of word problem: $bbbb \Rightarrow^*? aa$

- Convert to PCP

- $[bbbb^* ab \quad \underline{ab} \quad aa \quad \underline{aa} \quad a \quad \underline{a} \quad]$
 - $[\quad \underline{aba} \quad aba \quad \underline{a} \quad a \quad \underline{b} \quad b \quad \underline{*aa}]$
 - And $\begin{array}{ccccccc} * & * & a & \underline{a} & b & \underline{b} \\ * & * & \underline{a} & a & \underline{b} & b \\ - & & & & & \end{array}$

How PCP Construction Works?

- Using underscored letters avoids solutions that don't relate to word problem instance. E.g.,
ab aa
aba a
leads to solution no matter the question
- Top row insures start with $[W_0^*$
- Bottom row insures end with $_*W_f]$
- Bottom row matches W_i , while top matches W_{i+1} (one is underscored)
- Get Solution for PCP iff $W_0 \Rightarrow^* W_f$

Ambiguity of CFG

- Problem to determine if an arbitrary CFG is ambiguous

$$S \rightarrow A \mid B$$

$$A \rightarrow x_i A [i] \mid x_i [i] \quad 1 \leq i \leq n$$

$$B \rightarrow y_i B [i] \mid y_i [i] \quad 1 \leq i \leq n$$

$$A \Rightarrow^* x_{i_1} \dots x_{i_k} [i_k] \dots [i_1] \quad k > 0$$

$$B \Rightarrow^* y_{i_1} \dots y_{i_k} [i_k] \dots [i_1] \quad k > 0$$

- Ambiguous if and only if there is a solution to this PCP instance.

Intersection of CFLs

- Problem to determine if arbitrary CFG's define overlapping languages
- Just take the grammar consisting of all the A-rules from previous, and a second grammar consisting of all the B-rules. Call the languages generated by these grammars, L_A and L_B .
 $L_A \cap L_B \neq \emptyset$, if and only there is a solution to this PCP instance.

CSG Produces Something

$$S \rightarrow x_i S y_i^R \mid x_i T y_i^R \quad 1 \leq i \leq n$$

$$a T a \rightarrow * T *$$

$$* a \rightarrow a *$$

$$a * \rightarrow * a$$

$$T \rightarrow *$$

- Our only terminal is $*$. We get strings of form $*^{2j+1}$, for some j 's if and only if there is a solution to this PCP instance.

CSG Produces Something

- Our only terminal in previous grammar is $*$. We get strings of form $*^{2j+1}$, for some j 's if and only if there is a solution to this PCP instance. Get \emptyset otherwise.
- Thus, \mathbf{P} has a solution iff
 - $\mathbf{L(G)} \neq \emptyset$
 - $\mathbf{L(G)}$ is infinite

Traces and Grammars

Traces

- A valid trace
 - $\# C_1 \# C_2 \# C_3 \# C_4 \dots \# C_{k-1} \# C_k \#$,
where $k \geq 1$ and $C_i \Rightarrow_M C_{i+1}$, for $1 \leq i < k$.
Here, \Rightarrow_M means derive in **M**, and C is a valid ID (Instantaneous Description)
- An invalid trace
 - $\# C_1 \# C_2 \# C_3 \# C_4 \dots \# C_{k-1} \# C_k \#$,
where $k \geq 1$ and for some i , it is false that $C_i \Rightarrow_M C_{i+1}$.

Traces (Valid Computations)

- A terminating trace of a machine \mathbf{M} , is a word of the form
 $\# C_0 \# C_1 \# C_2 \# C_3 \# \dots \# C_{k-1} \# C_k \#$
where $C_i \Rightarrow C_{i+1}$, $0 \leq i < k$, C_0 is a starting configuration and C_k is a terminating configuration.
- We allow some laxness, where the configurations might be encoded in a manner appropriate to the machine model. Now, a context free grammar can be devised which approximates traces by either getting the even-odd pairs right, or the odd-even pairs right. The goal is to then intersect the two languages, so the result is a trace. This then allows us to create CFLs $\mathbf{L1}$ and $\mathbf{L2}$, where $\mathbf{L1} \cap \mathbf{L2} \neq \emptyset$, just in case the machine has an element in its domain. Since this is undecidable, the non-emptiness of the intersection problem is also undecidable. This is an alternate proof to one we already showed based on PCP.
- Additionally, if $\mathbf{L1} \cap \mathbf{L2} = \emptyset$, the complement (bad traces + non-traces) is Σ^* . As this can be shown to be a CFL, determining if a CFG generates Σ^* is undecidable as well.

What's Undecidable?

- We cannot decide if the set of valid terminating traces of an arbitrary machine **M** is non-empty.
- We cannot decide if the complement of the set of valid terminating traces of an arbitrary machine **M** is everything. In fact, this is not even semi-decidable.

What's a CSL or CFL?

- Given some machine **M** (I'll talk about specific models later)
 - The set of valid traces of **M** is Context Sensitive (can prove by fact that intersection of two CFLs is a CSG or by direct construction)
 - The complement of the valid traces of **M** is Context Free; that is, the set of invalid traces of **M** is Context Free (just one mistake required)
 - The set of valid terminating traces of **M** is Context Sensitive (same as above)
 - The complement of the valid terminating traces of **M** is Context Free; again, this requires just one mistake

$$L = \Sigma^*?$$

- If L is regular, then $L = \Sigma^*$? is decidable
 - Easy – Reduce to minimal deterministic FSA, \mathcal{A}_L accepting L . $L = \Sigma^*$ iff \mathcal{A}_L is a one-state machine, whose only state is accepting
- If L is context free, then $L = \Sigma^*$? is undecidable
 - Just produce the complement of a machine's valid terminating traces; if it's Σ^* then the original machine accepted nothing

Traces are NOT CFLs

- In the previous, we assumed that a trace is NOT a CFL, but we never proved that.
- To show the trace language for a TM, M ,
 $\{ \# C_1 \# C_2 \# C_3 \# C_4 \dots \# C_{k-1} \# C_k \# \mid$
 $k \geq 1 \text{ and } C_i \Rightarrow_M C_{i+1}, \text{ for } 1 \leq i < k \}$ is not a CFL, we can focus on a simple machine that has just one non-blank $\{1\}$ and one state $\{q\}$ and the rules
 $q 0 0 q$
 $q 1 1 q$
- This machine has traces of the form
 $\{ \# C \# C \# C \# C \dots \# C \# C \# \}$ as it never changes the tape contents or its state.

Using Pumping Lemma

- From previous slide, assume that the language of traces, $L = \{ \# C \# C \# C \# C \dots \# C \# C \# \}$, involving no changes in the ID is Context Free
- Pumping Lemma gives me an $N > 0$
- I choose the valid trace in L that is $\# q 1^N \# q 1^N \# q 1^N \#$
- PL breaks this up into $uvwxy$, $|vwx| \leq N$, $|vx| > 0$ and $\forall i \geq 0 uv^iwx^iy \in L$
- Case 1: vx contains some 1's. Due to fact that $|vwx| \leq N$, the 1's can come from at most two consecutive sequences of 1's. If $i=0$, then we reduce 1's in at most two subsequences, but not in the third, leading to an imbalance, and so the result is not in L .
- Case 2: vx contains no 1's, then it must be either 'q', '#', or '#q'. In any case, if $i=0$ then we remove a state or a divider or both and the result is not a sequence of fixed configurations, so is not in L .
- By PL, L is not a CFL.

Language of Traces is a CSL

- The easiest way to show this for Turing machine traces is to describe an LBA that is given a string and wants to check if it is a valid trace.
- The LBA could make a pass over to be sure the string starts with a #, ends with a #, has no 0's immediately following a #, has a leading 0 immediately prior to a # only if the character preceding that 0 is a state, and has exactly one state between each pair of #'s.
- The LBA could then check each pair by copying the second member of a pair under the first (2 tracks) and then marching over the two one character at a time until a state is found in one or the other. It can then do checks that are based on the Turing machine rules with there being a need to look at only 4 characters in each track – state, character to immediate left of state and up to two characters to immediate right of state on each track (think about it). Of course, all parts of configuration that are not altered must be checked to be sure they match on both tracks.

Non-Traces is a CFL

- There are two ways that a string might not be a valid trace.
- First, it might be ill-formed, but we can easily check if a word looks like a trace. If not, it is in the complement of valid traces
- Second, we can check pairs of configurations, $\# C_i \# C_{i+1}$ to see if there is a transcription error; that is, we can check to see if it is the case that C_{i+1} does not follow from C_i in a valid trace. This is a non-deterministic process where we “guess” which pair might be in error and then, if the guess is correct, we accept the string as a bad one that just looks like a trace.
- How hard is it to check for one bad transcription? Well, as noted above it starts with a guess, but then we must check. If it’s a TM trace, we use alternating ID reversals, so such a pair is either $\# C_i \# C_{i+1}^R$ or $\# C_i^R \# C_{i+1}$. Checking an error here is just looking as was described with the LBA single step check and can be done with a stack. What the stack cannot do is look at sequences longer than single pairs.

Traces of FRS with Residues

- I have chosen, once again to use the Factor Replacement Systems, but this time, Factor Systems with Residues. The rules are unordered and each is of the form $a x + b \rightarrow c x + d$

- These systems need to overcome the lack of ordering when simulating Register Machines. This is done by

$$\begin{array}{l}
 j. \quad \text{INC}_r[i] \quad p_{n+j} x \quad \rightarrow p_{n+i} p_r x \\
 j. \quad \text{DEC}_r[s, f] \quad p_{n+j} p_r x \quad \rightarrow p_{n+s} x \\
 \quad \quad \quad p_{n+j} p_r x + k p_{n+j} \quad \rightarrow p_{n+f} p_r x + k p_{n+f}, \quad 1 \leq k < p_r
 \end{array}$$

We also add the halting rule associated with $m+1$ of

$$p_{n+m+1} x \rightarrow 0$$

- Thus, halting is equivalent to producing 0. We can also add one more rule that guarantees we can reach 0 on both odd and even numbers of moves

$$0 \rightarrow 0$$

Intersection of CFLs

- Let $(n, ((a_1, b_1, c_1, d_1), \dots, (a_k, b_k, c_k, d_k)))$ be some factor replacement system with residues. Define grammars G_1 and G_2 by using the $4k+2$ rules

$$\begin{aligned} G : F_i &\rightarrow 1^{a_i} F_i 1^{c_i} \mid 1^{a_i+b_i} \# 1^{c_i+d_i} & 1 \leq i \leq k \\ S_1 &\rightarrow \# F_i S_1 \mid \# F_i \# & 1 \leq i \leq k \\ S_2 &\rightarrow \# 1^{x_0} S_1 1^{z_0} \# & Z_0 \text{ is 0 for us} \end{aligned}$$

G_1 starts with S_1 and G_2 with S_2

- Thus, using the notation of writing Y in place of 1^Y ,
 $L_1 = L(G_1) = \{ \# Y_0 \# Y_1 \# Y_2 \# Y_3 \# \dots \# Y_{2j} \# Y_{2j+1} \# \}$
 where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.

This checks the even/odd steps of an even length computation.

But, $L_2 = L(G_2) = \{ \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \dots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$
 where $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$.

This checks the odd/even steps of an even length computation.

- Given that the intersection of two CFLs is at worst a CSL, we now have an indirect way of showing that the valid terminating traces are a CSL.

Intersection Continued

Now, X_0 is chosen as some selected input value to the Factor System with Residues, and Z_0 is the unique value (0 in our case) on which the machine halts. But,

$$L1 \cap L2 = \{ \#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \dots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$$

where $X_i \Rightarrow X_{i+1}$, $0 \leq i < 2k$, and $X_{2k} \Rightarrow Z_0$. This checks all steps of an even length computation. But our original system halts if and only if it produces 0 (Z_0) in an even (also odd) number of steps. Thus the intersection is non-empty just in case the Factor System with residue eventually produces 0 when started on X_0 , just in case the Register Machine halts when started on the register contents encoded by X_0 .

This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.

What's a CSL or CFL?

- Given an FRS with Residue
 - The set of valid traces is Context Sensitive (can prove by fact that intersection of two CFLs is a CSG or by direct construction or by describing an LBA that accepts this language)
 - The set of valid traces is not Context Free (can use Pumping Lemma for this like earlier)
 - The complement of the valid traces is Context Free; that is, the set of invalid traces of M is Context Free (just one mistake required)
 - The set of valid terminating traces is Context Sensitive but not Context Free (same as above)
 - The complement of the valid terminating traces is Context Free; again, this requires just one mistake

Quotients of CFLs (concept)

Let $L1 = L(G1) = \{ \$ \# Y_0 \# Y_1 \# Y_2 \# Y_3 \# \dots \# Y_{2j} \# Y_{2j+1} \# \}$

where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.

This checks the even/odd steps of an even length computation.

Now, let $L2=L(G2)=\{X_0 \$ \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \dots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$

where $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$ and Z_0 is a unique halting configuration.

This checks the odd/steps of an even length computation and includes an extra copy of the starting number prior to its \$.

Now, consider the quotient of $L2 / L1$. The only way a member of $L1$ can match a final substring in $L2$ is to line up the \$ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting point (the one on which the machine halts.) Thus,

$L2 / L1 = \{ X_0 \mid \text{the system being traced halts} \}$.

Since deciding the members of an re set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.

Note: Intersection of two CFLs is a CSL but quotient of two CFLs is an re set and, in fact, all re sets can be specified by such quotients.

Quotients of CFLs (precise)

- Let $(n, ((a_1, b_1, c_1, d_1), \dots, (a_k, b_k, c_k, d_k)))$ be some factor replacement system with residues. Define grammars G_1 and G_2 by using the $4k+4$ rules

$$\begin{array}{llll}
 \mathbf{G} : \mathbf{F}_i & \rightarrow & \mathbf{1}^{a_i} \mathbf{F}_i \mathbf{1}^{c_i} \mid \mathbf{1}^{a_i+b_i} \# \mathbf{1}^{c_i+d_i} & \mathbf{1} \leq i \leq k \\
 \mathbf{T}_1 & \rightarrow & \# \mathbf{F}_i \mathbf{T}_1 \mid \# \mathbf{F}_i \# & \mathbf{1} \leq i \leq k \\
 \mathbf{A} & \rightarrow & \mathbf{1} \mathbf{A} \mathbf{1} \mid \mathbf{\$} \# & \\
 \mathbf{S}_1 & \rightarrow & \mathbf{\$} \mathbf{T}_1 & \\
 \mathbf{S}_2 & \rightarrow & \mathbf{A} \mathbf{T}_1 \# \mathbf{1}^{z_0} \# & \mathbf{Z}_0 \text{ is 0 for us}
 \end{array}$$

G_1 starts with S_1 and G_2 with S_2

- Thus, using the notation of writing Y in place of 1^Y ,
 $L_1 = L(G_1) = \{ \mathbf{\$} \# Y_0 \# Y_1 \# Y_2 \# Y_3 \# \dots \# Y_{2j} \# Y_{2j+1} \# \}$
 where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.

This checks the even/odd steps of an even length computation.

But, $L_2 = L(G_2) = \{ \mathbf{X} \mathbf{\$} \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \dots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$
 where $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$ and $X = X_0$

This checks the odd/steps of an even length computation, and includes an extra copy of the starting number prior to its $\mathbf{\$}$.

Summarizing Quotient

Now, consider the quotient $L2 / L1$ where $L1$ and $L2$ are the CFLs on prior slide. The only way a member of $L1$ can match a final substring in $L2$ is to line up the \$ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting number (the one on which the machine halts.) Thus,

$L2 / L1 = \{ X \mid \text{the system } F \text{ halts on zero} \}$.

Since deciding the members of an re set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.

Traces and Type 0

- Here, it is easier to show a simulation of a Turing machine than of an FRS.
- Assume we are given some machine M , with Turing table T (using Post notation). We assume a tape alphabet of Σ that includes a blank symbol B .
- Consider a starting configuration C_0 . Our rules will be

S	→	# C₀ #	where C₀ = $\alpha q_0 a \beta$ is initial ID
q a	→	s b	if $q a b s \in T$
b q a x	→	b a s x	if $q a R s \in T, a, b, x \in \Sigma$
b q a #	→	b a s B #	if $q a R s \in T, a, b \in \Sigma$
# q a x	→	# a s x	if $q a R s \in T, a, x \in \Sigma, a \neq B$
# q a #	→	# a s B #	if $q a R s \in T, a \in \Sigma, a \neq B$
# q a x	→	# s x #	if $q a R s \in T, x \in \Sigma, a = B$
# q a #	→	# s B #	if $q a R s \in T, a = B$
b q a x	→	s b a x	if $q a L s \in T, a, b, x \in \Sigma$
# q a x	→	# s B a x	if $q a L s \in T, a, x \in \Sigma$
b q a #	→	s b a #	if $q a L s \in T, a, b \in \Sigma, a \neq B$
# q a #	→	# s B a #	if $q a L s \in T, a \in \Sigma, a \neq B$
b q a #	→	s b #	if $q a L s \in T, b \in \Sigma, a = B$
# q a #	→	# s B #	if $q a L s \in T, a = B$
f	→	λ	if f is a final state
#	→	λ	just cleaning up the dirty linen

CSG and Undecidability

- We can almost do anything with a CSG that can be done with a Type 0 grammar. The only thing lacking is the ability to reduce lengths, but we can throw in a character that we think of as meaning “deleted”. Let’s use the letter d as a deleted character and use the letter e to mark both ends of a word.
- Let $G = (V, T, P, S)$ be an arbitrary Type 0 grammar.
- Define the CSG $G' = (V \cup \{S', D\}, T \cup \{d, e\}, S', P')$, where P' is

S'	\rightarrow	$e S e$	
$D x$	\rightarrow	$x D$	when $x \in V \cup T$
$D e$	\rightarrow	$e d$	push the delete characters to far right
α	\rightarrow	β	where $\alpha \rightarrow \beta \in P$ and $\alpha \leq \beta$
α	\rightarrow	βD^k	where $\alpha \rightarrow \beta \in P$ and $\alpha - \beta = k > 0$
- Clearly, $L(G') = \{ e w e d^m \mid w \in L(G) \text{ and } m \geq 0 \text{ is some integer} \}$
- For each $w \in L(G)$, we cannot, in general, determine for which values of m , $e w e d^m \in L(G')$. We would need to ask a potentially infinite number of questions of the form “does $e w e d^m \in L(G')$ ” for some $m \geq 0$ to determine if $w \in L(G)$. That’s a semi-decision procedure because m can be unbounded above.

Some Consequences

- CSGs are not closed under Init, Final, Mid, quotient with regular sets, substitution and homomorphism (okay for λ -free homomorphism and non-length reducing substitutions)
- We also have that the emptiness problem is undecidable from this result. That gives us two proofs of this one result.
- For Type 0, emptiness and even the membership problems are undecidable.

Undecidability

- Is $L = \emptyset$, for CSL, L? **PCP reduction**
- Is $L = \Sigma^*$, for CFL (CSL), L? **Trace Complement**
- Is $L_1 = L_2$ for CFLs (CSLs), L_1, L_2 ? **$L_1 = \Sigma^*$**
- Is $L_1 \subseteq L_2$ for CFLs (CSLs), L_1, L_2 ? **$L_1 = \Sigma^*$**
- Is $L_1 \cap L_2 = \emptyset$ for CFLs (CSLs), L_1, L_2 ? **PCP reduction**
- Is L regular, for CFL (CSL), L? **Think about it**
- Is $L_1 \cap L_2$ a CFL for CFLs, L_1, L_2 ? **Think about it**
- Is $\sim L$ CFL, for CFL, L? **Think about it**

More Undecidability

- Is CFL, L , ambiguous? **PCP**
- Is $L=L^2$, L a CFL? **Will Do**
- Is L_1/L_2 finite, L_1 and L_2 CFLs?
Language is any RE set
- Membership in L_1/L_2 , L_1 and L_2 CFLs?
Language is any RE set

Summary of Grammar Results

Decidability

- Everything about regular
- Membership in CFLs and CSLs
 - CKY for CFLs
- Emptiness for CFLs

Undecidability

- Is $L = \emptyset$, for CSL, L?
- Is $L = \Sigma^*$, for CFL (CSL), L?
- Is $L_1 = L_2$ for CFLs (CSLs), L_1, L_2 ?
- Is $L_1 \subseteq L_2$ for CFLs (CSLs), L_1, L_2 ?
- Is $L_1 \cap L_2 = \emptyset$ for CFLs (CSLs), L_1, L_2 ?

More Undecidability

- Is CFL, L , ambiguous?
- Is $L=L^2$, L a CFL?
- Does there exist a finite n , $L^n=L^{n+1}$?
- Is L_1/L_2 finite, L_1 and L_2 CFLs?
- Membership in L_1/L_2 , where L_1 and L_2 are CFLs?

Word to Grammar Problem

- Recast semi-Thue system making all symbols non-terminal, adding S and V to non-terminals and terminal set $\Sigma = \{a\}$

$$G: S \rightarrow h1^xq_10h$$

$$hq_0h \rightarrow V$$

$$V \rightarrow aV$$

$$V \rightarrow \lambda$$

- $x \in \mathcal{L}(M)$ iff $\mathcal{L}(G) \neq \emptyset$ iff $\mathcal{L}(G)$ infinite
iff $\lambda \in \mathcal{L}(G)$ iff $a \in \mathcal{L}(G)$ iff $\mathcal{L}(G) = \Sigma^*$

Consequences for PSG

- Unsolvables
 - $\mathcal{L}(G) = \emptyset$
 - $\mathcal{L}(G) = \Sigma^*$
 - $\mathcal{L}(G)$ infinite
 - $w \in \mathcal{L}(G)$, for arbitrary w
 - $\mathcal{L}(G) \supseteq \mathcal{L}(G2)$
 - $\mathcal{L}(G) = \mathcal{L}(G2)$
- Latter two results follow when have
 - $G2: S \rightarrow aS \mid \lambda \quad a \in \Sigma$

Finite Convergence for Concatenation of Context-Free Languages

Relation to Real-Time
(Constant Time) Execution

Powers of CFLs

Let G be a context free grammar.

Consider $L(G)^n$

Question1: Is $L(G) = L(G)^2$?

Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n > 0$?

These questions are both undecidable.

Think about why question1 is as hard as whether or not $L(G)$ is Σ^* .

Question2 requires much more thought.

$$L(G) = L(G)^2?$$

- **The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \bullet L \subseteq L$, so long as L is selected from a class of languages C over the alphabet Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.**
- **Corollary 1:**
The problem “is $L \bullet L = L$, for L context free or context sensitive?” is undecidable

$L(G) = L(G)^2?$ is undecidable

- **Question: Does $L \bullet L$ get us anything new?**
 - i.e., Is $L \bullet L = L$?
- **Membership in a CFL is decidable.**
- **Claim is that $L = \Sigma^*$ iff**
 - (1) $\Sigma \cup \{\lambda\} \subseteq L$; and
 - (2) $L \bullet L = L$
- **Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.**
- **Conversely, we have $\Sigma^* \subseteq L^* = \bigcup_{n \geq 0} L^n \subseteq L$**
 - first inclusion follows from (1); second from (2)

Finite Power Problem

- The problem to determine, for an arbitrary context free language L , if there exist a finite n such that $L^n = L^{n+1}$ is undecidable.
- $L_1 = \{ C_1 \# C_2^R \$ \mid C_1, C_2 \text{ are configurations} \}$,
- $L_2 = \{ C_1 \# C_2^R \$ C_3 \# C_4^R \dots \$ C_{2k-1} \# C_{2k}^R \$ \mid \text{where } k \geq 1 \text{ and, for some } i, 1 \leq i < 2k, C_i \Rightarrow_M C_{i+1} \text{ is false} \}$,
- $L = L_1 \cup L_2 \cup \{\lambda\}$.

Undecidability of $\exists n L^n = L^{n+1}$

- L is context free.
- Any product of L_1 and L_2 , which contains L_2 at least once, is L_2 . For instance, $L_1 \cdot L_2 = L_2 \cdot L_1 = L_2 \cdot L_2 = L_2$.
- This shows that $(L_1 \cup L_2)^n = L_1^n \cup L_2$.
- Thus, $L^n = \{\lambda\} \cup L_1 \cup L_1^2 \dots \cup L_1^n \cup L_2$.
- Analyzing L_1 and L_2 we see that $L_1^n \cup L_2 \neq L_2$ just in case there is a word $C_1 \# C_2^R \$ C_3 \# C_4^R \dots \$ C_{2n-1} \# C_{2n}^R \$$ in L_1^n that is not also in L_2 .
- But then there is some valid trace of length $2n$.
- L has the finite power property iff M executes in constant time.

Missing Step

- We have that **CT** (Constant-Time) is many-one reducible to Finite Power Problem (**FPC**) for CFLs
- This means that if **CT** is unsolvable, so is **FPC** for CFLs.
- However, we still lack a proof that **CT** is unsolvable. I am keeping that open as one of the problems that you folks can attack in your presentation. It takes two papers to get here. I'll document that.

Undecidability of Finite Convergence for Operators on Formal Languages

Relation to Real-Time
(Constant Time) Execution

Simple Operators

- Concatenation

- $A \bullet B = \{ xy \mid x \in A \ \& \ y \in B \}$

- Insertion

- $A \triangleright B = \{ xyz \mid y \in A, xz \in B, x, y, z \in \Sigma^* \}$

- Clearly, since x can be λ , $A \bullet B \subseteq A \triangleright B$

K-insertion

- $A \triangleright^{[k]} B = \{ x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \mid$
 $y_1 y_2 \dots y_k \in A,$
 $x_1 x_2 \dots x_k x_{k+1} \in B,$
 $x_i, y_j \in \Sigma^* \}$
- Clearly, $A \bullet B \subseteq A \triangleright^{[k]} B$, for all $k > 0$

Iterated Insertion

- $A^{(1)} \triangleright^{[n]} B = A \triangleright^{[n]} B$
- $A^{(k+1)} \triangleright^{[n]} B = A \triangleright^{[n]} (A^{(k)} \triangleright^{[n]} B)$

Shuffle

- Shuffle (product and bounded product)
 - $A \diamond B = \cup_{j \geq 1} A \triangleright^{[j]} B$
 - $A \diamond^{[k]} B = \cup_{1 \leq j \leq k} A \triangleright^{[j]} B = A \triangleright^{[k]} B$
- One is tempted to define shuffle product as $A \diamond B = A \triangleright^{[k]} B$ where
$$k = \mu y [A \triangleright^{[j]} B = A \triangleright^{[j+1]} B]$$
but such a k may not exist – in fact, we will show the undecidability of determining whether or not k exists

More Shuffles

- Iterated shuffle

- $A \diamond^0 B = A$

- $A \diamond^{k+1} B = (A \diamond^{[k]} B) \diamond B$

- Shuffle closure

- $A \diamond^* B = \cup_{k \geq 0} (A \diamond^{[k]} B)$



Crossover

- Unconstrained crossover is defined by
 $A \otimes_u B = \{ wz, yx \mid wx \in A \text{ and } yz \in B \}$

- Constrained crossover is defined by
 $A \otimes_c B = \{ wz, yx \mid wx \in A \text{ and } yz \in B, \\ |w| = |y|, |x| = |z| \}$

Who Cares?

- People with no real life (me?)
- Insertion and a related deletion operation are used in biomolecular computing and dynamical systems
- Shuffle is used in analyzing concurrency as the arbitrary interleaving of parallel events
- Crossover is used in genetic algorithms

Some Known Results

- Regular languages, A and B
 - $A \bullet B$ is regular
 - $A \triangleright^{[k]} B$ is regular, for all $k > 0$
 - $A \diamond B$ is regular
 - $A \diamond^* B$ is not necessarily regular
 - Deciding whether or not $A \diamond^* B$ is regular is an open problem

More Known Stuff

- CFLs, A and B
 - $A \bullet B$ is a CFL
 - $A \triangleright B$ is a CFL
 - $A \triangleright^{[k]} B$ is not necessarily a CFL, for $k > 1$
 - Consider $A = a^n b^n$; $B = c^m d^m$ and $k = 2$
 - Trick is to consider $(A \triangleright^{[2]} B) \cap a^* c^* b^* d^*$
 - $A \diamond B$ is not necessarily a CFL
 - $A \diamond^* B$ is not necessarily a CFL
 - Deciding whether or not $A \diamond^* B$ is a CFL is an open problem

Immediate Convergence

- $L = L^2$?
- $L = L \triangleright L$?
- $L = L \diamond L$?
- $L = L \diamond^* L$?
- $L = L \otimes_c L$?
- $L = L \otimes_u L$?

Finite Convergence

- $\exists k > 0 \ L^k = L^{k+1}$
 - $\exists k \geq 0 \ L(k) \triangleright L = L(k+1) \triangleright L$
 - $\exists k \geq 0 \ L \triangleright^{[k]} L = L \triangleright^{[k+1]} L$
 - $\exists k \geq 0 \ L \diamond^k L = L \diamond^{k+1} L$
 - $\exists k \geq 0 \ L(k) \otimes_c L = L(k+1) \otimes_c L$
 - $\exists k \geq 0 \ L(k) \otimes_u L = L(k+1) \otimes_u L$
-
- $\exists k \geq 0 \ A(k) \triangleright B = A(k+1) \triangleright B$
 - $\exists k \geq 0 \ A \triangleright^{[k]} B = A \triangleright^{[k+1]} B$
 - $\exists k \geq 0 \ A \diamond^k B = A \diamond^{k+1} B$
 - $\exists k \geq 0 \ A(k) \otimes_c B = A(k+1) \otimes_c B$
 - $\exists k \geq 0 \ A(k) \otimes_u B = A(k+1) \otimes_u L$

Finite Power of CFG

- Let G be a context free grammar.
- Consider $L(G)^n$
- Question1: Is $L(G) = L(G)^2$?
- Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n > 0$?
- These questions are both undecidable.
- Think about why question1 is as hard as whether or not $L(G)$ is Σ^* .
- Question2 requires much more thought.

1981 Results

- Theorem 1:
The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \bullet L \subseteq L$, so long as L is selected from a class of languages C over the alphabet Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.
- Corollary 1:
The problem “is $L \bullet L = L$, for L context free or context sensitive?” is undecidable

Proof #1

- Question: Does $L \bullet L$ get us anything new?
 - i.e., Is $L \bullet L = L$?
- Membership in a CSL is decidable.
- Claim is that $L = \Sigma^*$ iff
 - (1) $\Sigma \cup \{\lambda\} \subseteq L$; and
 - (2) $L \bullet L = L$
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \bigcup_{n \geq 0} L^n \subseteq L$
 - first inclusion follows from (1); second from (2)

Subsuming •

- Let \oplus be any operation that subsumes concatenation, that is $A \bullet B \subseteq A \oplus B$.
 - Simple insertion is such an operation, since $A \bullet B \subseteq A \triangleright B$.
 - Unconstrained crossover also subsumes
 - ,
- $$A \otimes_c B = \{ wz, yx \mid wx \in A \text{ and } yz \in B \}$$

$$L = L \oplus L ?$$

- Theorem 2:
The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \oplus L \subseteq L$, so long as $L \bullet L \subseteq L \oplus L$ and L is selected from a class of languages C over Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.

Proof #2

- Question: Does $L \oplus L$ get us anything new?
 - i.e., Is $L \oplus L = L$?
- Membership in a CSL is decidable.
- Claim is that $L = \Sigma^*$ iff
 - (1) $\Sigma \cup \{\lambda\} \subseteq L$; and
 - (2) $L \oplus L = L$
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \bigcup_{n \geq 0} L^n \subseteq L$
 - first inclusion follows from (1); second from (1), (2) and the fact that $L \bullet L \subseteq L \oplus L$

Propositional Calculus

Axiomatizable Fragments

Propositional Calculus

- Mathematical of unquantified logical expressions
- Essentially Boolean algebra
- Goal is to reason about propositions
- Often interested in determining
 - Is a well-formed formula (wff) a tautology?
 - Is a wff refutable (unsatisfiable)?
 - Is a wff satisfiable? (classic NP-complete)

Tautology and Satisfiability

- The classic approaches are:
 - Truth Table
 - Axiomatic System (axioms and inferences)
- Truth Table
 - Clearly exponential in number of variables
- Axiomatic Systems Rules of Inference
 - Substitution and Modus Ponens
 - Resolution / Unification

Proving Consequences

- Start with a set of axioms (all tautologies)
- Using substitution and MP
($P, P \supset Q \Rightarrow Q$)
derive consequences of axioms (also tautologies, but just a fragment of all)
- Can create complete sets of axioms
- Need 3 variables for associativity, e.g.,
($p1 \vee p2$) \vee $p3 \supset p1 \vee (p2 \vee p3)$

Some Undecidables

- Given a set of axioms,
 - Is this set complete?
 - Given a tautology T , is T a consequent?
- The above are even undecidable with one axiom and with only 2 variables. I will show this result shortly.

Refutation

- If we wish to prove that some wff, F , is a tautology, we could negate it and try to prove that the new formula is refutable (cannot be satisfied; contains a logical contradiction).
- This is often done using resolution.

Resolution

- Put formula in Conjunctive Normal Form (CNF)
- If have terms of conjunction $(P \vee Q)$, $(R \vee \sim Q)$ then can determine that $(P \vee R)$
- If we ever get a null conclusion, we have refuted the proposition
- Resolution is not complete for derivation, but it is for refutation

Axioms

- Must be tautologies
- Can be incomplete
- Might have limitations on them and on WFFs, e.g.,
 - Just implication
 - Only n variables
 - Single axiom

Simulating Machines

- Linear representations require associativity, unless all operations can be performed on prefix only (or suffix only)
- Prefix and suffix-based operations are single stacks and limit us to CFLs
- Can simulate Post normal Forms with just 3 variables.

Diadic PIPC

- Diadic limits us to two variables
- PIPC means Partial Implicational Propositional Calculus, and limits us to implication as only connective
- Partial just means we get a fragment
- Problems
 - Is fragment complete?
 - Can F be derived by substitution and MP?

Living without Associativity

- Consider a two-stack model of a TM
- Could somehow use one variable for left stack and other for right
- Must find a way to encode a sequence as a composition of forms – that's the key to this simulation

Composition Encoding

- Consider $(p \supset p)$, $(p \supset (p \supset p))$,
 $(p \supset (p \supset (p \supset p)))$, ...
 - No form is a substitution instance of any of the other, so they can't be confused
 - All are tautologies
- Consider $((X \supset Y) \supset Y)$
 - This is just $X \vee Y$

Encoding

- Use $(p \supset p)$ as form of bottom of stack
- Use $(p \supset (p \supset p))$ as form for letter 0
- Use $(p \supset (p \supset (p \supset p)))$ as form for 1
- Etc.
- String 01 (reading top to bottom of stack) is
 - $(((p \supset p) \supset ((p \supset p) \supset ((p \supset p) \supset (p \supset p)))) \supset (((p \supset p) \supset ((p \supset p) \supset ((p \supset p) \supset (p \supset p)))) \supset ((p \supset p) \supset ((p \supset p) \supset ((p \supset p) \supset (p \supset p)))))))$

Encoding

$I(p)$ abbreviates $[p \supset p]$

$\Phi_0(p)$ is $[p \supset I(p)]$ which is $[p \supset [p \supset p]]$

$\Phi_1(p)$ is $[p \supset \Phi_0(p)]$

$\xi_1(p)$ is $[p \supset \Phi_1(p)]$

$\xi_2(p)$ is $[p \supset \xi_1(p)]$

$\xi_3(p)$ is $[p \supset \xi_2(p)]$

$\psi_1(p)$ is $[p \supset \xi_3(p)]$

$\psi_2(p)$ is $[p \supset \psi_1(p)]$

...

$\psi_m(p)$ is $[p \supset \psi_{m-1}(p)]$

Creating Terminal IDs

1. $[\xi_1 I(p_1) \vee I(p_1)]$.
2. $[\xi_1 I(p_1) \vee I(p_1)] \supset [\xi_1 I(p_1) \vee \Phi_1 I(p_1)]$.
3. $[\xi_1 I(p_1) \vee \Phi_i(p_2)] \supset [\xi_1 I(p_1) \vee \Phi_j \Phi_i(p_2)], \forall i, j \in \{0, 1\}$.
4. $[\xi_1 I(p_1) \vee p_2] \supset [\xi_2 \Phi_1 I(p_1) \vee p_2]$.
5. $[\xi_1 I(p_1) \vee p_2] \supset [\xi_3 \Phi_i I(p_1) \vee p_2], \forall i \in \{0, 1\}$.
6. $[\xi_2 \Phi_i(p_1) \vee p_2] \supset [\xi_2 \Phi_j \Phi_i(p_1) \vee p_2], \forall i, j \in \{0, 1\}$.
7. $[\xi_2 \Phi_i(p_1) \vee p_2] \supset [\xi_3 \Phi_j \Phi_i(p_1) \vee p_2], \forall i, j \in \{0, 1\}$.
8. $[\xi_3 \Phi_i(p_1) \vee p_2] \supset [\Psi_k \Phi_i(p_1) \vee p_2]$, whenever $q_k a_i$ is a terminal discriminant of M .

Reversing Print and Left

9. $[\Psi_k \Phi_i(p_1) \vee p_2] \supset [\Psi_h \Phi_j(p_1) \vee p_2]$, whenever $q_h a_j a_i q_k \in T$.
- 10a. $[\Psi_k \Phi_0 I(p_1) \vee I(p_1)] \supset [\Psi_h \Phi_0 I(p_1) \vee I(p_1)]$,
b. $[\Psi_k \Phi_1 I(p_1) \vee I(p_1)] \supset [\Psi_h \Phi_0 I(p_1) \vee \Phi_1(p_1)]$,
c. $[\Psi_k \Phi_i I(p_1) \vee \Phi_j(p_2)] \supset [\Psi_h \Phi_0 I(p_1) \vee \Phi_i \Phi_j(p_2)]$,
d. $[\Psi_k \Phi_0 \Phi_0 \Phi_i(p_1) \vee I(p_2)] \supset [\Psi_h \Phi_0 \Phi_i(p_1) \vee I(p_2)]$,
e. $[\Psi_k \Phi_1 \Phi_0 \Phi_i(p_1) \vee I(p_2)] \supset [\Psi_h \Phi_0 \Phi_i(p_1) \vee \Phi_1 I(p_2)]$,
f. $[\Psi_k \Phi_i \Phi_0 \Phi_j(p_1) \vee \Phi_m(p_2)] \supset [\Psi_h \Phi_0 \Phi_j(p_1) \vee \Phi_i \Phi_m(p_2)]$,
 $\forall i, j, m \in \{0, 1\}$ whenever $q_h 0 L q_k \in T$.
- 11a. $[\Psi_k \Phi_0 \Phi_1(p_1) \vee I(p_2)] \supset [\Psi_h \Phi_1(p_1) \vee I(p_2)]$,
b. $[\Psi_k \Phi_1 \Phi_1(p_1) \vee I(p_2)] \supset [\Psi_h \Phi_1(p_1) \vee \Phi_1 I(p_2)]$,
c. $[\Psi_k \Phi_i \Phi_1(p_1) \vee \Phi_j(p_2)] \supset [\Psi_h \Phi_1(p_1) \vee \Phi_i \Phi_j(p_2)]$,
 $\forall i, j \in \{0, 1\}$ whenever $q_k 1 L q_k \in T$.

Reversing Right

- 12a. $[\Psi_k \Phi_0 I(p_1) \vee I(p_1)] \supset [\Psi_h \Phi_0 I(p_1) \vee I(p_1)],$
b. $[\Psi_k \Phi_0 I(p_1) \vee \Phi_0 \Phi_i(p_2)] \supset [\Psi_h \Phi_0 I(p_1) \vee \Phi_i(p_2)],$
c. $[\Psi_k \Phi_1(p_1) \vee I(p_2)] \supset [\Psi_h \Phi_0 \Phi_1(p_1) \vee I(p_2)],$
d. $[\Psi_k \Phi_0 \Phi_i(p_1) \vee I(p_2)] \supset [\Psi_h \Phi_0 \Phi_0 \Phi_i(p_1) \vee I(p_2)],$
e. $[\Psi_k \Phi_0 \Phi_i(p_1) \vee \Phi_0 \Phi_j(p_2)] \supset [\Psi_h \Phi_0 \Phi_0 \Phi_i(p_1) \vee \Phi_j(p_2)],$
f. $[\Psi_k \Phi_1(p_1) \vee \Phi_0 \Phi_i(p_2)] \supset [\Psi_h \Phi_0 \Phi_1(p_1) \vee \Phi_i(p_2)],$
 $\forall i, j \in \{0, 1\}$ whenever $q_h 0 R q_k \in T.$
- 13a. $[\Psi_k \Phi_0 I(p_1) \vee \Phi_1(p_2)] \supset [\Psi_h \Phi_1 I(p_1) \vee p_2],$
b. $[\Psi_k \Phi_1(p_1) \vee \Phi_1(p_2)] \supset [\Psi_h \Phi_1 \Phi_1(p_1) \vee p_2],$
c. $[\Psi_k \Phi_0 \Phi_i(p_1) \vee \Phi_1(p_2)] \supset [\Psi_h \Phi_1 \Phi_0 \Phi_i(p_1) \vee p_2]$
 $\forall i \in \{0, 1\}$ whenever $q_h 1 R q_k \in T.$

Exam Prep

Sample Question

Let **A** and **B** be re sets. For each of the following, either prove that the set is re, or give a counterexample that results in some known non-re set.

Let A be semi decided by f_A and B by f_B

a) $A \cup B$: must be re as it is semi-decided by

$$f_{A \cup B}(x) = \exists t [\text{stp}(f_A, x, t) \parallel \text{stp}(f_B, x, t)]$$

b) $A \cap B$: must be re as it is semi-decided by

$$f_{A \cap B}(x) = \exists t [\text{stp}(f_A, x, t) \&\& \text{stp}(f_B, x, t)]$$

c) $\sim A$: can be non-re. If $\sim A$ is always re, then all re are recursive as any set that is re and whose complement is re is decidable. However, $A = K$ is a non-rec, re set and so $\sim A$ is not re.

Sample Question

Given that the predicate **STP** and the function **VALUE** are prf's, show that we can semi-decide

{ f | φ_f evaluates to 0 for some input }

This can be shown re by the predicate

{ f | $\exists \langle x, t \rangle [stp(f, x, t) \ \&\& \ value(f, x, t) = 0] \}$

Sample Question

Let S be an re (recursively enumerable), non-recursive set, and T be re, non-empty, possibly recursive set. Let $E = \{ z \mid z = x + y, \text{ where } x \in S \text{ and } y \in T \}$.

(a) Can E be non re? **No as we can let S and T be semi-decided by f_S and f_T , resp., E is then semi-dec. by**

$f_E(z) = \exists \langle x, y, t \rangle [\text{stp}(f_S, x, t) \ \&\& \ \text{stp}(f_T, y, t) \ \&\& \ (z = \text{value}(f_S, x, t) + \text{value}(f_T, y, t))]$

(b) Can E be re non-recursive? **Yes, just let $T = \{0\}$, then $E = S$ which is known to be re, non-rec.**

(c) Can E be recursive? **Yes, let $T = \mathbb{N}$, then $E = \{ x \mid x \geq \min(S) \}$ which is a co-finite set and hence rec.**

Sample Question

Assuming **TOTAL** is undecidable, use reduction to show the undecidability of **Incr** = $\{ f \mid \forall x \varphi_f(x+1) > \varphi_f(x) \}$

Let f be arb.

Define $G_f(x) = \varphi_f(x) - \varphi_f(x) + x$

$f \in \text{TOTAL}$ iff $\forall x \varphi_f(x) \downarrow$ iff $\forall x G_f(x) \downarrow$ iff

$\forall x \varphi_f(x) - \varphi_f(x) + x = x$ iff $G_f \in \text{Incr}$

Sample Question

Let $\text{Incr} = \{ f \mid \forall x, \varphi_f(x+1) > \varphi_f(x) \}$.

Let $\text{TOTAL} = \{ f \mid \forall x, \varphi_f(x) \downarrow \}$.

Prove that $\text{Incr} \leq_m \text{TOTAL}$.

Let f be arb.

Define $G_f(x) = \exists t[\text{stp}(f,x,t) \ \&\& \ \text{stp}(f,x+1,t) \ \&\& \ (\text{value}(f,x+1,t) > \text{value}(f,x,t))]$

$f \in \text{Incr}$ iff $\forall x \varphi_f(x+1) > \varphi_f(x)$ iff
 $\forall x G_f(x) \downarrow$ iff $G_f \in \text{TOT}$

Sample Question

Let $\text{Incr} = \{ f \mid \forall x \varphi_f(x+1) > \varphi_f(x) \}$.

Use Rice's theorem to show Incr is not recursive.

Non-Trivial as

$C_0(x)=0 \notin \text{Incr}; S(x)=x+1 \in \text{Incr}$

Let f, g be arb. Such that $\forall x \varphi_f(x) = \varphi_g(x)$

$f \in \text{Incr}$ iff $\forall x \varphi_f(x+1) > \varphi_f(x)$ iff

$\forall x \varphi_g(x+1) > \varphi_g(x)$ iff $g \in \text{Incr}$

Sample Question

Let S be a recursive (decidable set), what can we say about the complexity (recursive, re non-recursive, non-re) of T , where $T \subset S$?

Nothing. Just let $S = \mathbb{N}$, then T could be any subset of \mathbb{N} . There are an uncountable number of such subsets and some are clearly in each of the categories above.

Sample Question

Let $P = \{ f \mid \exists x [\text{STP}(f, x, x)] \}$. Why does Rice's theorem not tell us anything about the undecidability of P ?

This is not an I/O property as we can have implementations of C_0 that are efficient and satisfy P and others that do not.