University of Central Florida

Complexity Theory Computability

Charles E. Hughes

COT6410 – Spring 2022 Notes

Slides whose headers are italicized red are not required material

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.

HISTORY

The Quest for Mechanizing Mathematics

Hilbert, Russell and Whitehead

- Until 1800's there were no formal systems to reason about mathematical properties
- Major advances in late 1800's/early 1900's
- Axiomatic schemes
  - Axioms plus sound rules of inference
  - Much of focus on number theory
- First Order Predicate Calculus
  - $\forall x \exists y\ [y > x]$
- Second Order (Peano's Axiom)
  - $\forall P\ [[P(0)\ \&\&\ \forall x[P(x) \Rightarrow P(x+1)]] \Rightarrow \forall x P(x)]$

Hilbert

- 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 zeros of polynomial equations with integral coefficients. This is now known to be impossible (In 1970, Matiyacevič showed this undecidable).
- Davis based on prior work by Julia Robinson, him and Hilary Putnam provided more readable proof in 1972.

Hilbert's Belief

- All mathematics could be developed within a formal system that allowed the mechanical creation and checking of proofs.

Gödel

- In 1931 he showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- He did this by showing that such a first order theory cannot reason about itself. That is, there is a first order expressible proposition that cannot be either proved or disproved, or the theory is inconsistent (some proposition and its complement are both provable).
- Gödel also developed the general notion of recursive functions but made no claims about their strength.

Turing (Post, Church, Kleene)

- In 1936, each presented a formalism for computability.
  – Turing and Post devised abstract machines and claimed these represented all mechanically computable functions.
  – Church developed the notion of lambda-computability from recursive functions (as previously defined by Gödel and Kleene) and claimed completeness for this model.
- Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.
- Church’s notation was the lambda calculus, which later gave birth to Lisp.

More on Emil Post

- In the 1920's, starting with notation developed by Frege and others in 1880s, Post devised the truth table form we all use now for Boolean expressions (propositional logic). This was a part of his PhD thesis in which he showed the axiomatic completeness of the propositional calculus (all tautologies can be deduced from a finite set of tautologies and a finite set of rules of inference – substitution and modus ponens).
- In the late 1930's and the 1940's, Post devised symbol manipulation systems in the form of rewriting rules (precursors to Chomsky's grammars). He showed their equivalence to Turing machines.
- In 1940s, Post showed the undecidability)of determining what is derivable from an arbitrary set of propositional axioms.

Terminology and Context

Notation and Vocabulary

Procedures vs Algorithms

- Recall that Algorithms always Halt but Procedures Might Not for Some Input
- *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**. **Ø** 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 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 = Ø) or there exists an algorithm F, over the natural numbers ***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

- **P** enumerable iff **P** semi-decidable.
- **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 10^{th}

- 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 10^{th} is Semi-Decidable

- Consider over one variable: $P(x) = 0$
- Can semi-decide by plugging in
  0, 1, -1, 2, -2, 3, -3, …
- 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 $P(x_1,x_2,..,x_k) = 0$.

P(x) = 0 is Decidable

- $c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0 = 0$
- $x^n = -(c_{n-1} x^{n-1} + \ldots + c_1 x + c_0)/c_n$
- $|x^n| \leq c_{max}(|x^{n-1}| + \ldots + |x| + 1|)/|c_n|$
- $|x^n| \leq c_{max}(n\ |x^{n-1}|)/|c_n|$, 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 [± 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 $P(x_1,x_2,..,x_k) = 0$.

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 $\{y \mid f(x)\downarrow \text{ and } f(x) = y\}$.

More terminology

We typically discuss procedures over the Natural Numbers.

The Greek symbol mu, **µ**, is read “the least value.”

Specifically, **µ y [P(y)]**, is read “the least of **y** such that **P(y)** is true.

If **P(y)** is never true, then **f(x) = µ y [P(y)]** always diverges.

A simple example is **↑(x) = µ y [y = y+1]**. Note ↑ as procedure name

If **P(y)** is true somewhere then we have an algorithm.

We often have a second argument here as in **g(x) = µ y [P(x,y)]**.

The **domain** of **g** is the set of values, **x**, for which **g(x)**↓.

The **range** of **g** is the union of the set of values it produces for all input.

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, denoted **Univ** or sometimes φ.
This program gets two inputs. The first is a description of the program to be simulated and the second is 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. The list of programs is $\varphi_0, \varphi_1, \varphi_2, \cdots$

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

We might use any of these notations

Halting Problem

Given procedure index **x** and input **y**, does φ_x **(y)** $\downarrow$

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

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

Now we can view Halt as a mapping from $\aleph$ into $\aleph$ by treating its input as a single number representing the pairing of two numbers via the one-one onto function

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

with inverses

$$<z>_1 = \log_2(z+1)$$

$$<z>_2 = (((z + 1) // 2^{<z>_1}) - 1) // 2$$

Picture Proof

φ/x	0	1	.	.	.	x	.	.
φ_0	$\varphi_0(0)\downarrow$?	$\varphi_0(1)\downarrow$?				$\varphi_0(x)\downarrow$?		
φ_1	$\varphi_1(0)\downarrow$?	$\varphi_1(1)\downarrow$?				$\varphi_1(x)\downarrow$?		
.								
.								
.								
φ_x	$\varphi_x(0)\downarrow$?	$\varphi_x(1)\downarrow$?				$\varphi_x(x)\downarrow$?		
.								
.								

If **Halt** exist we can create this "lazy" evaluation.
Consider diagonal and a procedure defined by
Disagree(x) = μy [not Halt(x,x)].
So **Disagree(x)=0 iff φ_x(x)**$\uparrow$ and **Disagree(x)**$\uparrow$ **iff φ_x(x)**$\downarrow$
If **Halt** exists then so does **Disagree** and it is the **d**-th row for some **d**.
But then **φ_d(d)**$\downarrow$ **iff not Halt(d,d) iff φ_d(d)**$\uparrow$

The Contradiction in Text

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

$$\textbf{Disagree}(x) = \begin{cases} \mathbf{0} & \text{if } \textbf{not Halt}(x,x)\text{, i.e, if } \varphi_x(x)\uparrow \\ \mu y\,[\textbf{ not Halt}(x,x)\,] & \\ \uparrow & \text{if } \textbf{Halt}(x,x)\text{, i.e, if } \varphi_x(x)\downarrow \end{cases}$$

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

Disagree(d) is defined ⇔ **not Halt(d,d)**

⇔ $\varphi_d(d)\uparrow$

⇔ **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 wrong.

Thus, **Halt** does not exist and the **Halting Problem** is not solvable.

Halting is recognizable

While the **Halting Problem** is not solvable, it is re, 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";
}
```

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

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

  where $\mathbf{A_0}$, $\mathbf{A_1}$, $\mathbf{A_2}$, … is a list of indexes of all and only the algorithms

Picture Proof

F/x	0	1	.	.	.	x	.	.
A_0	$A_0(0)$	$A_0(1)$				$A_0(x)$		
A_1	$A_1(0)$	$A_1(1)$				$A_1(x)$		
.								
.								
.								
A_x	$A_x(0)$	$A_x(1)$				$A_x(x)$		
.								
.								

If **F** exist we can create this “lazy” evaluation.
Consider diagonal and an algorithm **G** defined by
G(x) = Univ(F(x) , x) + 1 = $\varphi_{F(x)}(x) = A_x(x) + 1$
If **F** exists then so does **G** and it is the **g**-th row for some **g**.
But then **G(g) =** $A_g(g) + 1 = G(g) + 1$
As **G** is an algorithm, this is a contradiction, So **F** does not exist.

The Contradiction in Text

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

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

  Then, $\mathbf{G(g) = A_g(g) + 1 = G(g) + 1}$
- But then **G** contradicts its own existence since **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 **G(g)** is undefined. In fact, we already have shown how to enumerate the procedures, also known as 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 for 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 or repeat loops are needed, even if implicit (recursion) 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 well-behaved 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 $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$
- Q is finite set of states
- Σ, is a finite input alphabet not containing the blank symbol ⊔
- Γ is finite set of tape symbols that includes Σ and ⊔ commonly $\Gamma = \Sigma \cup \{⊔\}$
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R,L\}$
- 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 present
- 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.

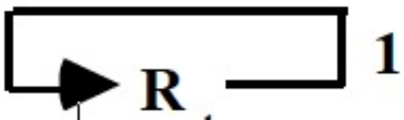
Base Machines

- 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

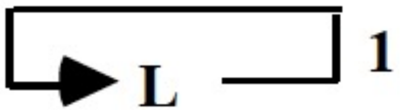
$\mathcal{R}$ -- move right to next 0 (not including current square)

$\ldots\underline{?}11\ldots10\ldots \Rightarrow \ldots?11\ldots1\underline{0}\ldots$



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

$\ldots011\ldots1\underline{?}\ldots \Rightarrow \ldots\underline{0}11\ldots1?\ldots$



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, x1, … , xn in the form

$$\ldots 01^{x1}01^{x2}0\ldots 01^{xn}\underline{0}\ldots$$

and ends with

$$\ldots 01^{x1}01^{x2}0\ldots 01^{xn}01^{y}\underline{0}\ldots$$

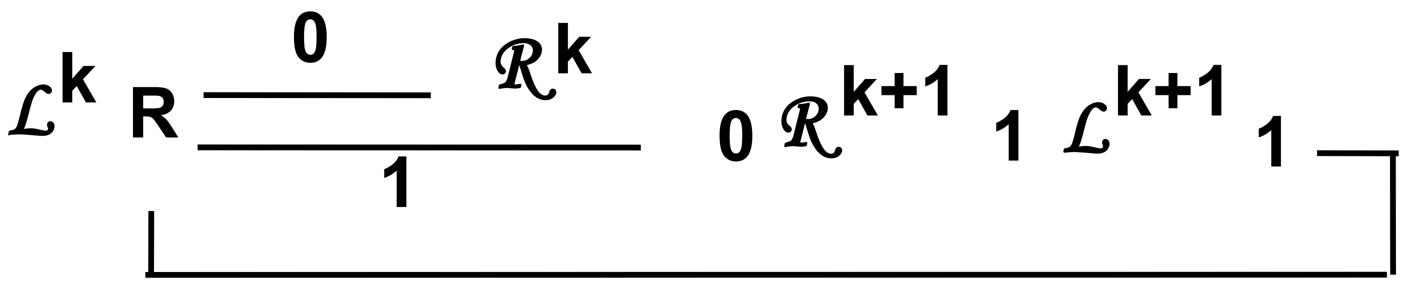
where **y = F(x1, … , xn)**

Why matters – Composition, e.g.,

y = G(x1, … , xn, F(x1, … , xn))

Addition by TM

Need the copy family of useful submachines, where $\mathbf{C_k}$ copies $\mathbf{k}$-th preceding value.



The add machine is then

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

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:

  $\mathbf{INC_r[i]}$ –

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

  $\mathbf{DEC_r[p, z]}$ –

  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 ← r1 - r2, if r1≥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 **x**.
- A fraction **a/b** is applicable to some natural number **x**, just in case **x** is divisible by **b**. We always chose the first applicable fraction (**a/b**), multiplying it times **x** to produce a new natural number **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 **n**-ary function **F** typically starts with its argument encoded as powers of the first **n** odd primes. Thus, arguments **x1**,**x2**,…,**xn** are encoded as $3^{x1}5^{x2}\ldots p_n^{xn}$. The result then appears as the power of the prime **2**.

Addition by FRS

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

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

2 / 3

2 / 5

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

$\mathbf{bx} \to \mathbf{ax}$

meaning that a number that has can be factored as **bx** can have the factor **b** replaced by an **a**.

The previous rules would then be written

$\mathbf{3x} \to \mathbf{2x}$

$\mathbf{5x} \to \mathbf{2x}$

Limited Subtraction by FRS

Subtraction is $\mathbf{3^{x1}5^{x2}}$ becomes $\mathbf{2^{max(0,x1-x2)}}$

$$3 \cdot 5x \rightarrow x$$
$$3x \rightarrow 2x$$
$$5x \rightarrow x$$

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

$$\mathbf{3{\cdot}5x \rightarrow x}$$
$$\mathbf{3x \rightarrow 2x}$$
$$\mathbf{5x \rightarrow x}$$

Starting with $\mathbf{135 = 3^3 5^1}$, deterministically we get

$$\mathbf{135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$$

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

$$\mathbf{135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$$
$$\mathbf{135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3}$$
$$\mathbf{135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1}$$
$$\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0}$$
$$\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0}$$
$$\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1}$$
$$\mathbf{135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$$
$$\mathbf{135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3}$$

…

This computes $\mathbf{2^z}$ where $\mathbf{0 \leq z \leq x_1}$. 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

$$\mathbf{2x} \rightarrow \mathbf{x}$$

$$\mathbf{2x} \rightarrow \mathbf{4x}$$

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 → x

3 x → 2 x

Sample FRS

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

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

$\mathbf{3*5*7\ x \rightarrow x}$

$\mathbf{3*5\ x \rightarrow 2\ x}$

$\mathbf{5*7\ x \rightarrow 7*11\ x}$

$\mathbf{7*11\ x \rightarrow 3*11\ x}$

$\mathbf{11\ x \rightarrow 5\ x}$

$\mathbf{5\ x \rightarrow x}$

$\mathbf{7\ x \rightarrow x}$

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 of 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.

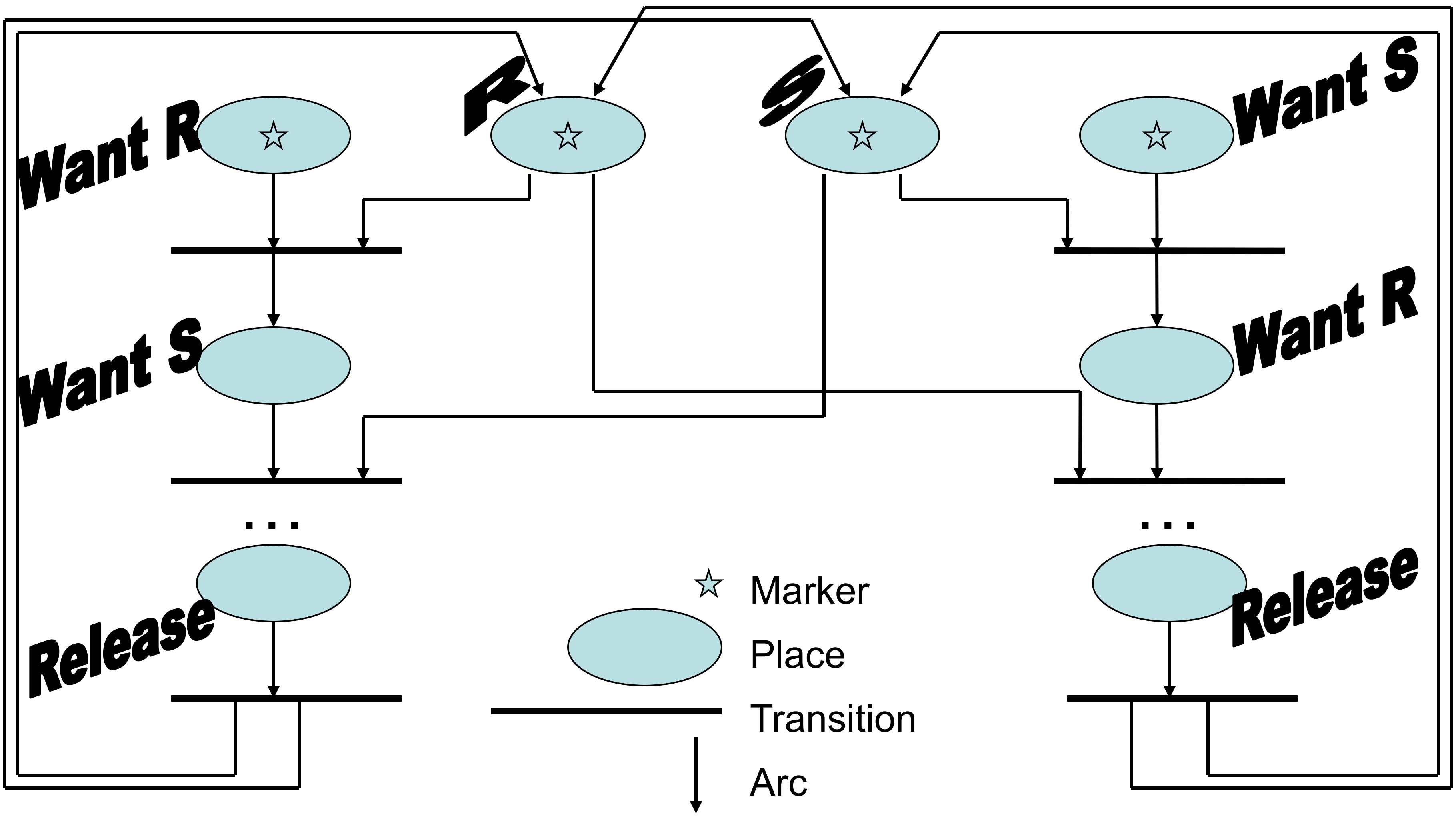
Petri Net Computation

- 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., **<0,1,4,0,6>** 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 transitions 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
  - $\mathbf{a_i\ x + c_i \ \rightarrow \ b_i\ x + d_i}$
  - There are **n** such rules
  - Can apply if number is such that you get a residue (remainder) $\mathbf{c_i}$ when you divide by $\mathbf{a_i}$
  - Take quotient **x** and produce a new number $\mathbf{b_i\ x + d_i}$
  - Can apply any applicable one (no order)
- These systems are equivalent to Register Machines.

Abelian Semi-Group

S = (G, •) is a semi-group if

G is a set, • is a binary operator, and

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

S is a monoid if

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

S is a group if

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

S is Abelian if • is commutative

Finitely Presented

- **S = (G, •)**, 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 **G**.
- Note, the set **G** is the closure of the generators under the semi-group's operator **•**.
- 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:

$\mathbf{C_a(x_1,\ldots,x_n) = a}$	: constant functions
$\mathbf{I^n_i(x_1,\ldots,x_n) = x_i}$	: identity functions
	: aka projection
$\mathbf{S(x) = x+1}$	: an increment function

Building New Functions

- **Composition:**

  If **G**, $\mathbf{H_1}$, … , $\mathbf{H_k}$ are already known to be primitive recursive, then so is **F**, where

  $$\mathbf{F(x_1,\ldots,x_n) = G(H_1(x_1,\ldots,x_n), \ldots , H_k(x_1,\ldots,x_n))}$$

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

  If **G**, **H** are already known to be primitive recursive, then so is **F**, where

  $$\mathbf{F(0, x_1,\ldots,x_n) = G(x_1,\ldots,x_n)}$$

  $$\mathbf{F(y+1, x_1,\ldots,x_n) = H(y, x_1,\ldots,x_n, F(y, x_1,\ldots,x_n))}$$

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

Addition & Multiplication

Example: Addition

$+(0,y) = I^1_1(y)$

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

where $H(a,b,c) = S(I^3_3(a,b,c))$

Example: Multiplication

$*(0,y) = C_0(y)$

$*(x+1,y) = H(x,y,*(x,y))$

where $H(a,b,c) = +(I^3_2(a,b,c), I^3_3(a,b,c))$

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

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 **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) =** $\mathbf{C_0}$)
  x*(y+1) = +(x,x*y) (more formally ***(x,y+1) = +(x,*(x,y))**)

Basic Arithmetic

x + 1: // aka S(x) – successor

x + 1 = S(x)

x – 1: // aka P(x) – predecessor

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:

$x * 0 = 0$

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

x!:

$0! = 1$

$(x+1)! = (x+1) * x!$

Basic Relations

x == 0:

0 == 0 = 1

(y+1) == 0 = 0

x == y:

x==y = ((x – y) + (y – x)) == 0

x ≤y :

x≤y = (x – y) == 0

x ≥ y:

x≥y = y≤x

x > y :

x>y = ~(x≤y) /* See ~ on next page */

x < y :

x<y = ~(x≥y)

Basic Boolean Operations

~x:

~x = 1 – x or (x==0)

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

~(x==0)

x && y:

x&&y = signum(x*y)

x || y:

x||y = ~((x==0) && (y==0))

Definition by Cases

One case

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

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

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

$$f(x) = \begin{cases} g_1(x) & \text{if } P_1(x) \\ g_2(x) & \text{if } P_2(x) \text{ \&\& } \sim P_1(x) \\ \ldots & \\ g_k(x) & \text{if } P_k(x) \text{ \&\& } \sim(P_1(x) \mid\mid \ldots \mid\mid \sim P_{k-1}(x)) \\ h(x) & \text{otherwise} \end{cases}$$

Bounded Minimization 1

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

Can show **f** is primitive recursive by

$\mathbf{f(0)}$	$=$	$\mathbf{1-P(0)}$	
$\mathbf{f(x+1)}$	$=$	$\mathbf{f(x)}$	if $\mathbf{f(x) \le x}$
	$=$	$\mathbf{x+2-P(x+1)}$	otherwise

Bounded Minimization 2

f(x) = μ z (z < x) [P(z)] if ∃ such a **z,**
= x, otherwise
where **P(z)** is primitive recursive.

Can show **f** is primitive recursive by
f(0) = 0
f(x+1) = μ z (z ≤ x) [P(z)]

Intermediate Arithmetic

x // y:

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

x//(y+1) = μ 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**.

firstfactor(x) = $\mu z \ (2 \le z \le x) \ [\, z|x \,]$,

0 if none

isPrime(x):

isPrime(x) = firstFactor(x) == x && (x>1)

prime(i) = i-th prime:

prime(0) = 2

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

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

Exponents

x^y:

x^0 = 1

x^(y+1) = x * x^y

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

exp(x,i) = μ z (z<x) [~(p_i^(z+1) | x)]

Pairing Functions

- **pair(x,y) = $\langle x,y \rangle = 2^x \, (2y + 1) - 1$**
- with inverses

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

  $\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$
- These are very useful and can be extended to encode **n**-tuples

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

Pairing Function is 1-1 Onto

Prove that the pairing function **<x,y> = 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, **<x,y>+1**, which implies the problem of mapping the pairing function to $\mathbf{Z^+}$.

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 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**, **<x,y>+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^xz**, where **x>0**, **z** is an odd number and this pair **x,y** is unique. Thus, **<x,y>+1** is **1-1** onto the even natural numbers, when x>0.

The above shows that **<x,y>+1** is **1-1** onto **Z^+**, but then **<x,y>** is **1-1** onto $\aleph$, 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
$\mathbf{f^{-1}(f(x)) = x}$, for arbitrary $\mathbf{x}\in\mathbf{S}$ and that
$\mathbf{f(f^{-1}(z)) = z}$, for arbitrary $\mathbf{z}\in\mathbf{T}$.

Thus, we need to show that
$\mathbf{(\langle x,y\rangle_1,\langle x,y\rangle_2) = (x,y)}$ for arbitrary $\mathbf{(x,y)}\in\aleph\times\aleph$ and
$\mathbf{\langle\langle z\rangle_1,\langle z\rangle_2\rangle = z}$ for arbitrary $\mathbf{z}\in\aleph$.

Alternate Proof

Let **x,y** be arbitrary natural number, then $\mathbf{<x,y> = 2^x(2y+1)-1}$.

Moreover, $\mathbf{<2^x(2y+1)-1>_1 = Factor(2^x(2y+1),0) = x}$, since **2y+1** must be odd, and

$\mathbf{<2^x(2y+1)-1>_2 = ((2^x(2y+1)/2\text{^}Factor(2^x(2y+1),0))-1)/2 = 2y/2 = y}$.

Thus, $\mathbf{(<x,y>_1,<x,y>_2) = (x,y)}$, as was desired.

Let **z** be an arbitrary natural number, then the inverse of the pairing is $\mathbf{(<z>_1,<z>_2)}$

Moreover, $\mathbf{<<z>_1,<z>_2> = 2\text{^}<z>_1 *(2<z>_2+1)-1}$

= 2^Factor(z+1,0)*(2*((z+1)/ 2^Factor(z+1,0))/2-1+1)-1

= 2^Factor(z+1,0)*((z+1)/ 2^Factor(z+1,0))-1

= (z+1) – 1

= z, as was desired.

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); 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.

$\mathbf{K(0,x) = B(x)}$

$\mathbf{B(x) = < g(x), C_0(x) >}$ // this is just $\mathbf{<g(x), 0>}$

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

$\mathbf{J(z) = < h(<z>_1, <z>_2), <z>_1 >}$

// this is $\mathbf{< f(y+1,x), f(y,x)>}$, even though **f** is not yet shown to be prf!!

This shows **K** is prf.

f is then defined from **K** as follows:

$\mathbf{f(y,x) = <K(y,x)>_1}$ // extract first value from pair encoded in $\mathbf{K(y,x)}$

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

- **$A(1, j)=2j$ 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 **G** is already known to be recursive, then so is **F**, where

  $$\mathbf{F(x1,\ldots,xn) = \mu y\ (G(y,x1,\ldots,xn) == 1)}$$

- 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, $\mathbf{id_i} \Rightarrow_M \mathbf{id_{i+1}}$, $\mathbf{i \geq 0}$, if the execution of a single instruction of ***M*** would alter ***M***'s state from $\mathbf{id_i}$ to $\mathbf{id_{i+1}}$ or if ***M*** halts in state $\mathbf{id_i}$ and $\mathbf{id_{i+1}=id_i}$.
- $\Rightarrow^+_M$ is the transitive closure of $\Rightarrow_M$
- $\Rightarrow^*_M$ is the reflexive transitive closure of $\Rightarrow_M$

id Definitions

- For a register machine, **M**, an id is an **s+1** tuple of the form $\mathbf{(i, r_1,\ldots,r_s)_M}$ 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, **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 $\alpha\mathbf{qx}\beta$, where α (β) is the shortest string representing all non-blank squares to the left (right) of the scanned square, **x** is the symbol at the scanned square and **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, **h**, from id's of ***M*** into those of ***M'***, such that, $\mathbf{R}_M$ **=** { **h(d) | d** is an instance of an id of ***M*** }, and
  - $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{h(id_0)}$, and $\mathbf{h(id_0)}$ is the only member of $\mathbf{R}_M$ in the configurations encountered in this derivation.
  - $\mathbf{h(id_i)} \Rightarrow^+_{M'} \mathbf{h(id_{i+1})}$, $\mathbf{i \geq 0}$, and $\mathbf{h(id_{i+1})}$ 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 $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{h(id)}$, and
  - If $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{id'}$ then either $\mathbf{id_0} \Rightarrow^*_M \mathbf{id}$, where $\mathbf{id' = h(id)}$, 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 ≤ REGISTER ≤ FACTOR ≤
  RECURSIVE ≤ TURING
  where, by **A ≤ 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 ≤ REGISTER

Encoding a TM's State

- Assume that we have an **n** state Turing machine. Let the states be numbered **0,…, n-1**.
- Assume our machine is in state **7**, with its tape containing
  … 0 0 1 0 1 0 0 1 1 q7 <u>0</u> 0 0 …
- The underscore indicates the square being read. We denote this by the finite id
  1 0 1 0 0 1 1 q7 <u>0</u>
- 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 <u>0</u> 0 0 …
  is just (**0, 83, 7**).
  … 0 0 1 0 q5 <u>1</u> 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 MUL_r2__BY_2 JUMP[set_k]	: If (qj 0 R qk) is rule in TM
qj+1.	MUL_r1_BY_2 IF_r2_ODD then INC1 DIV_r2__BY_2[set_k]	: If (qj 0 L qk) is rule in TM
…		
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…10$_2$, where, #1's = r1**
2.	**DEC1[3,4]**	**: r1 will be set to 0**
3.	**INCn+1[1]**	**:**
4.	**MUL_rn+1_BY_2[5]**	**: Set rn+1 = 11…1011…10$_2$, where, #1's = r1, then r2**
5.	**DEC2[6,7]**	**: r2 will be set to 0**
6.	**INCn+1[4]**	**:**
…		
3n-2.	**DECn[3n-1,3n+1]**	**: Set rn+1 = 11…1011…1011…1$_2$, where, #1's = r1, r2,…**
3n-1.	**MUL_rn+1_BY_2[3n]**	**: rn will be set to 0**
3n.	**INCn+1[3n-2]**	**:**
3n+1	**DECn+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 $\mathbf{p_i}$'s are distinct primes and the $\mathbf{k_i}$'s are non-zero values, except that the number **1** would be represented by $\mathbf{2^0}$.
- Let R be an arbitrary **n+1**-register machine, having m instructions.

  Encode the contents of registers **r0,…,rn** by the powers of $\mathbf{p_0,\ldots p_n}$.

  Encode rule number's **1,…,m** by primes $\mathbf{p_{n+1} ,\ldots, p_{n+m}}$

  Use $\mathbf{p_{n+m+1}}$ 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≤j≤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 → x

3 x → 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 $\mathbf{3^x 5^y 7}$

7 • 5 x	$\rightarrow$	**11 x**
7 x	$\rightarrow$	**13 x**
11 • 3 x	$\rightarrow$	**7 x**
11 x	$\rightarrow$	**7 x**
13 • 3 x	$\rightarrow$	**17 x**
13 x	$\rightarrow$	**19 x**
17 x	$\rightarrow$	**13 • 2 x**
19 x	$\rightarrow$	**x**

Analysis of Problem

- If we don't obey the ordering here, we could take an input like **$3^5 5^2 7$** and immediately apply the second rule (the one that mimics a failed decrement).
- We then have **$3^5 5^2 13$**, signifying that we will mimic instruction number **3**, never having subtracted the **2** from **5**.
- Now, we mimic copying **r1** to **r0** and get **$2^5 5^2 19$** .
- 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$_2$), … ,(a$_n$,b$_n$))** be some factor replacement system, where **(a$_i$,b$_i$)** means that the **i**-th rule is

  a$_i$x → b$_i$x

- Encode this machine by the number **F**,

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

Simulation by Recursive # 1

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

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

- Note: $\mathbf{exp(F,2{*}i{-}1) = a_i}$ 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, $\mathbf{1 \le i \le n}$, for which this is true, then **RULE(F,x) = i**.

  If x is not divisible by any $\mathbf{a_i}$, $\mathbf{1 \le i \le n}$, then **x** is divisible by **1**, and **RULE(F,x)** returns **n+1**. That's why we added $\mathbf{p_{2n+1}\ p_{2n+2}}$.

- Given the function **RULE(F,x)**, we can determine **NEXT(F,x)**, the number that follows **x**, when using **F**, by

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

Simulation by Recursive # 2

- The configurations listed by **F**, when started on **x**, are

CONFIG(F, x, 0) = x

CONFIG(F, x, y+1) = NEXT(F, CONFIG(F, x, y))

- The number of the configuration on which **F** halts is

HALT(F, x) = μ y [CONFIG(F, x, y) == 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

  Univ(F, x) = exp (CONFIG (F, x, HALT (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

- $2^0 3^a 5^b \Rightarrow 2^{a-b}$
  $3*5x \rightarrow x$ or 1/15
  $5x \rightarrow x$ or 1/5
  $3x \rightarrow 2x$ 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) = μ z $(1 \le z \le 4)$ [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$(F, 3^4\ 5^2) = (3^4\ 5^2 \mathbin{//} 15 * 1) = 3^3 5^1$
  NEXT$(F, 3^3\ 5^1) = (3^3\ 5^1 \mathbin{//} 15 * 1) = 3^2$
  NEXT$(F, 3^2) = (3^2 \mathbin{//} 3 * 2) = 2^1 3^1$
  NEXT$(F, 2^1 3^1) = (2^1 3^1 \mathbin{//} 3 * 2) = 2^2$
  NEXT$(F, 2^2) = (2^2 \mathbin{//} 1 * 1) = 2^2$

Rest of simulation

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

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

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, **x1, … , xn** in the form

  $\ldots 01^{x1}01^{x2}0\ldots01^{xn}\underline{0}\ldots$

  and ends with

  $\ldots 01^{x1}01^{x2}0\ldots01^{xn}01^{y}\underline{0}\ldots$

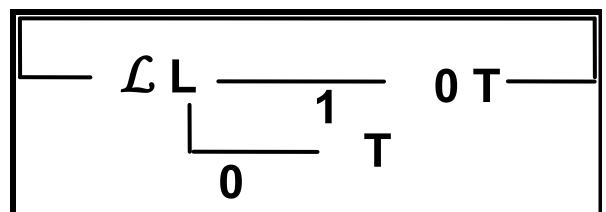
  where **y = F(x1, … , xn)**.

More Helpers

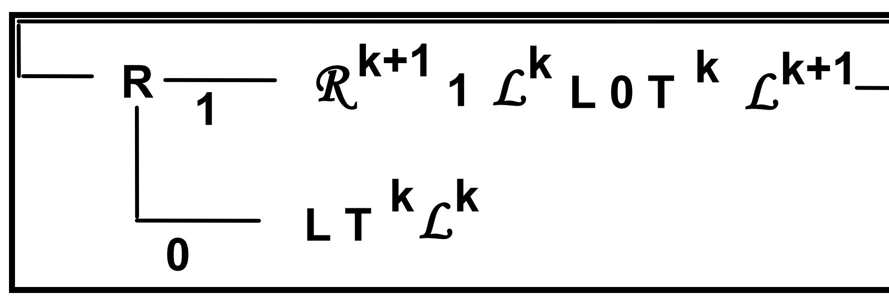
- To build our simulation we need to construct some useful submachines, in addition to the $\mathcal{R}$, $\mathcal{L}$, **R**, **L**, and $\mathbf{C_k}$ machines already defined.
- **T** -- translate moves a value left one tape square
  $\dots\underline{?}01^x0\dots \Rightarrow \dots?1^x\underline{0}0\dots$

$\boxed{\mathbf{R1}\mathcal{R}\mathbf{L0}}$

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



- $\mathbf{Rot_k}$ -- Rotate a **k** value sequence one slot to the left
  $\dots\underline{0}1^{x1}01^{x2}0\dots01^{xk}0\dots$
  $\Rightarrow \dots\underline{0}1^{x2}0\dots01^{xk}01^{x1}0\dots$



Basic Functions

All Basis Recursive Functions are Turing computable:

- $\mathbf{C_a^n(x_1,\ldots,x_n) = a}$

  $\mathbf{(R1)^aR}$

- $\mathbf{I_i^n\ (x_1,\ldots,x_n) = x_i}$

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

- $\mathbf{S(x) = x+1}$

  $\mathbf{C_1 1R}$

Closure Under Composition

If **$G, H_1, \ldots, H_k$** are already known to be Turing computable, then so is **F**, where

$$\mathbf{F(x_1,\ldots,x_n) = G(H1(x_1,\ldots,x_n), \ldots , Hk(x_1,\ldots,x_n))}$$

To see this, we must first show that if **$E(x_1,\ldots,x_n)$** is Turing computable then so is

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

This can be computed by the machine

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

Can now define **F** by

$$\mathbf{H_1\ H_2\langle 1\rangle\ H_3\langle 2\rangle \ldots H_k\langle k\text{-}1\rangle\ G\ Shift^k}$$

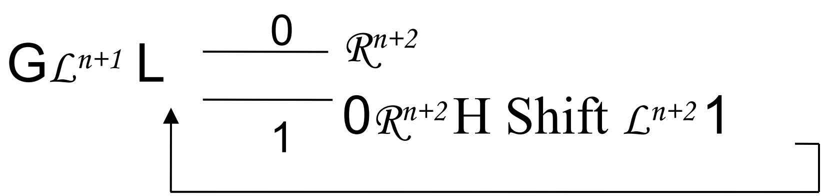
Closure Under Induction

To prove that Turing Machines are closed under induction (primitive recursion), we must simulate some arbitrary primitive recursive function **$F(y,x_1,x_2, \ldots, x_n)$** on a Turing Machine, where

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

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

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



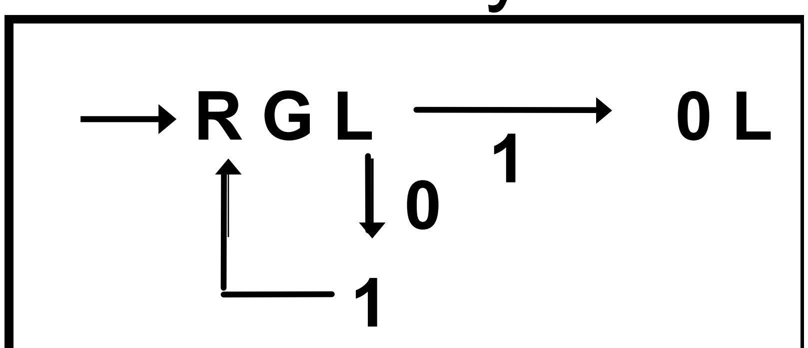
Since our Turing Machine simulator can produce the same value for any arbitrary PRF, **F**, we show that Turing Machines are closed under induction (primitive recursion).

Closure Under Minimization

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

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

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. **$| \Sigma^* | = |\aleph|$**, where $\aleph$ 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, **$| L | \leq | \Sigma^* | = | \aleph |$**)

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, $\mathbf{F_P}$: $\Sigma_I^* \rightarrow \Sigma_O^*$ where Σ_I is the input alphabet and Σ_O is the output alphabet.
8. $\mathbf{F_P}$ defines an input language $\mathbf{P_I}$ for which $\mathbf{F_P}$ 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 $\mathbf{F_P}$, $\mathbf{P_O}$, is the set of outputs. That is, $\mathbf{P_O = \{ y \mid \exists x}$ in $\mathbf{P_I}$ and $\mathbf{y = F_P (x) \}}$
9. Since there are a countable number of programs, **P**, there can be at most a countable number of functions $\mathbf{F_P}$ and consequently, only a countable number of distinct input languages and output languages associated with programs in $\mathbf{L_P}$. 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.

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 projection/substitution.

Universal Machine Mapping

- $\varphi^{(n)}(f, x_1,\ldots,x_n) = \text{Univ}\ (f, \prod_{i=1}^{n} p_i^{x_i})$
- We will sometimes adopt the above and also its common shorthand

  $\varphi_f^{(n)}(x_1,\ldots,x_n) = \varphi^{(n)}(f, x_1,\ldots,x_n)$

  and the even shorter version

  $\varphi_f(x_1,\ldots,x_n) = \varphi^{(n)}(f, x_1,\ldots,x_n)$

SNAP and TERM

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

  SNAP(f, x, t) = CONFIG(f, x, t)
- Termination in our notation occurs when we reach a fixed point, so

  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, x1,…,xn, t)** is a predicate defined to be true iff φ_f **(x1,…,xn)** converges in at most **t** steps.
- **STP** is primitive recursive since it can be defined by

  STP(f, x, *t*) = TERM(f, 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 φ_f **(x1,…,xn)** so long as
  STP(f, x1,…,xn, t) is true.
- **VALUE(f, x1,…,xn, t) =**
  exp (CONFIG (F, x, t), 0), where
  x is an encoiding of **<x1,…,xn>**
- **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.

  S $\subseteq \aleph$ is semi-decidable iff there exists a partially computable function **g** where

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

- I prefer the definition of re that says
  S $\subseteq \aleph$ is re iff **S =** $\emptyset$ or there exists a totally computable function f where

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

- We will prove these equivalent. Actually, **f** can be a primitive recursive function.

Semi-Decidable Implies re

Theorem: Let **S** be semi-decided by $\mathbf{G_S}$. Assume $\mathbf{G_S}$ is the $\mathbf{g_S}$–th function in our enumeration of effective procedures. If **S = Ø** then **S** is re by definition, so we will assume wlog that there is some ***a*** ∈ **S**. Define the enumerating algorithm $\mathbf{F_S}$ by

$$\mathbf{F_S(\langle x,t\rangle) = x * STP(g_s, x, t) + \mathit{a} * (1\text{-}STP(g_s, x, t))}$$

Note: $\mathbf{F_S}$ is <u>primitive recursive</u> and it enumerates every value in **S** infinitely often.

re Implies Semi-Decidable

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

$$\psi_S(x) = \begin{cases} \mu y\,[y == y+1] & \text{if } S == \varnothing \\ \exists y[F_S(y)==x], & \text{otherwise} \end{cases}$$

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

Domain of a Procedure

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

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

$$\psi_S(x) = \begin{cases} \mu y\,[y == y+1] & \text{if } S == \emptyset \\ x * \exists y[F_S(y)==x], & \text{otherwise} \end{cases}$$

Recursive Implies re

Theorem: Recursive implies re.

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

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

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

Clearly

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

Related Results

Theorem: **S** is re iff **S** is semi-decidable.

Proof: That's what we proved.

Theorem: **S** and **~S** are both re (semi-decidable) iff **S** (equivalently **~S**) is recursive (decidable).

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

$$\mathbf{g_S(x) = STP(f_S,\ x,\ \mu t\ (STP(f_S,\ x,\ t)\ ||\ STP(f_{S'}\ ,x,\ t))}$$

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

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

Enumeration Theorem

- Define
  $$\mathbf{W_n = \{ x \in \aleph \mid \varphi(n,x)\downarrow \}}$$
- Theorem: A set **B** is re iff there exists an **n** such that $\mathbf{B = W_n}$.
  Proof: Follows from definition of $\varphi\mathbf{(n,x)}$.
- This gives us a way to enumerate the recursively enumerable sets.
- Note: We cannot enumerate the recursive sets as they are non-re.

The Set K

- $\mathbf{K = \{ n \in \aleph \mid n \in W_n \}}$
- Note that
  $\mathbf{n \in W_n \Leftrightarrow \varphi(n,n)\downarrow \Leftrightarrow HALT(n,n)}$
- 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 **~K** is not re.
- If **~K** is re then **~K = W$_{\mathbf{i}}$**, for some **i**.
- However, this is a contradiction since
  $\mathbf{i} \in \mathbf{K} \Leftrightarrow \mathbf{i} \in \mathbf{W_i} \Leftrightarrow \mathbf{i} \in \mathbf{\sim K} \Leftrightarrow \mathbf{i} \notin \mathbf{K}$

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 $\mathbf{f_S}$ enumerates S.
   $\mathbf{g_S(x) = x * STP(f_S, x, \mu t\ (STP(f_S, x, t))}$ or
   $\mathbf{g_S(x) = x * \exists t\ STP(f_S, x, t)}$

S-m-n Theorem

Parameter (S-m-n) Theorem

- Theorem: For each **n,m>0**, there is a prf $\mathbf{S_m^n(y, u_1,\ldots,u_n)}$ such that

  $$\varphi^{(m+n)}(y, x_1,\ldots,x_m, u_1,\ldots,u_n)$$
  $$= \varphi^{(m)}(S_m^n(y,u_1,\ldots,u_n), x_1,\ldots, x_m)$$
- 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 $\mathbf{u_i}$ as the exponent of the prime $\mathbf{p_{m+i}}$.
- Sketch of proof:
  Assume we normally start with $\mathbf{p_1^{x1} \ldots p_m^{xm}\ p_1^{u1} \ldots p_{m+n}^{un}}\ \sigma$
  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 $\mathbf{p_1^{x1} \ldots p_m^{xm}}$, with **q** the first unused prime.

  $\mathbf{q}\ \alpha\ \mathbf{x} \rightarrow \mathbf{q}\ \beta\ \mathbf{x}$ — replaces $\alpha\ \mathbf{x} \rightarrow \beta\ \mathbf{x}$ in **F**, for each rule in **F**
  $\mathbf{q\ x} \rightarrow \mathbf{q\ x}$ — ensures we loop at end
  $\mathbf{x} \rightarrow \mathbf{q\ p_{m+1}^{u1} \ldots p_{m+n}^{un}}\ \sigma\ \mathbf{x}$
  adds fixed input, start state and **q**
  this is selected once and never again

  Note: **q = prime(max(n+m, lastFactor(Product[i=1 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 $\mathbf{2^r 3^{a_1} 5^{b_1} \ldots p_{2r-1}^{a_r} p_{2r}^{b_r}}$
- $\mathbf{S_{m,n}(F, u_1,\ldots u_n) = 2^{r+2} 3^{q\times a_1} 5^{q\times b_1} \ldots p_{2r-1}^{q\times a_r} p_{2r}^{q\times b_r}}$
  $\mathbf{p_{2r+1}^{q} p_{2r+2}^{q}\, p_{2r+3} p_{2r+4}^{\ q}\, p_{m+1}^{u_1} \ldots p_{m+n}^{u_n}\, \sigma}$
- 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 $\mathbf{u_1,\ldots u_n}$, 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.
- $\mathbf{S_{m,n} = S_m^{\,n}}$ is clearly primitive recursive. I'll leave the precise proof of that as a challenge to you.

Quantification

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
  $\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists \mathbf{t}\ \mathbf{A_S(x,t)}$
  This is clear since, if $\mathbf{g_S}$ is the index of the procedure ψ_S that semi-decides **S**, then
  $\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists \mathbf{t}\ \mathbf{STP(g_S, x, t)}$
  So, $\mathbf{A_S(x,t) = STP_{g_S}(\ x, t\ )}$, 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 $\mathbf{A_S}$ such that
 $\mathbf{x \notin S \Leftrightarrow \forall t\ A_S(x,t)}$
 This is clear since, if g_S is the index of the procedure ψ_S that semi-decides **S**, then
 $\mathbf{x \notin S \Leftrightarrow \sim\exists t\ STP(g_S, x, t) \Leftrightarrow \forall t \sim STP(g_S, x, t)}$
 So, $\mathbf{A_S(x,t) = \sim STP_{g_S}(x, t)}$, 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.

Quantification#3

- The **Uniform Halting Problem** was already shown to be non-re. It turns out its complement is also not re. In fact, we can show that **TOT** requires an alternation of quantifiers as it is neither re nor co-re. Specifically,

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

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

Comments About Set of Algorithms

The Set TOT

- The listing of all algorithms can be viewed as
  TOT = { f ∈ ℵ | ∀x φ(f, x)↓ }

- We can also note that
  TOT = { f ∈ ℵ | W_f =ℵ }

- 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.

Reducibility

Reduction Concepts

- Proofs by contradiction are tedious after you've seen a few (we saw three–Halt, TOTAL, and ~K). 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, **S**, is undecidable, we can show it is as least as hard as the set **HALT = K_0**. That is, **$K_0 \le S$**. 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, **S**, is co-re, non-recursive, we can show it is the complement of some re, non-recursive set.
- To show a set, **S**, is not re and not even co-re, we can show it is as least as hard as the set **TOTAL (the set of algorithms)**. That is, **TOTAL $\le$ S**. We can also do this by showing it is the complement of a non-re, non-co-re set.

Some Precise Upper Bounds

- **$<f,x> \in$ HALT $\Leftrightarrow \exists t$ [STP(f,x,t)]**
  Hence re upper bound
- **$<f,x> \in$ NON_HALT $\Leftrightarrow \forall t$ [~STP(f,x,t)]**
  Hence co-re upper bound

- **f $\in$ TOTAL $\Leftrightarrow \forall x\ \exists t$ [STP(f,x,t)]**
  Non-re, Non-co-re upper bound
- **f $\in$ NON_TOTAL $\Leftrightarrow \exists\ x\ \forall t$ [~STP(f,x,t)]**
  Non-re, Non-co-re upper bound

Reduction Example#1

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

Reduction Examples #2 & #3

In all cases below we are assuming our variables are over $\aleph$.
HALT = { <f,x> | φ_f (x)↓ } is unsolvable (undecidable, non-recursive)
TOTAL = { f | ∀x φ_f (x)↓ } = { f | W_f = $\aleph$ } is not even recursively enumerable (re, semidecidable)
Consider ZERO = { f | ∀x φ_f (x) = 0 }
f∈ZERO⇔∀x∃t[STP(f,x,t)&&VALUE(f,x,t)==0] gives upper bound

- Show **ZERO = { f | ∀x φ_f (x) = 0 }** is unsolvable – Show lower bound
  <f,x> ∈ **HALT iff g(y) = φ_f (x) - φ_f (x)** is zero for all **y**.
  Thus, **<f,x>** ∈ **HALT** iff **g** ∈ **ZERO** (really the index of **g**).
  A solution to **ZERO** implies one for **HALT**, so **ZERO** is unsolvable.
- Show **ZERO = { f | ∀x φ_f (x) = 0 }** is non-re – Attain upper bound
  f ∈ **TOTAL** iff **h(x) = φ_f (x) - φ_f (x)** is zero for all **x**.
  Thus, **f** ∈ **TOTAL** iff **h** ∈ **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
  HALT = K_0 = L_u = { <f, x> | φ_f (x) is defined }
- Membership problem for **L_u** is the **Halting Problem**.

- The sets **L_{ne}** and **L_e**, where

  NON-EMPTY = L_{ne} = { f | $\exists$ x φ_f (x) $\downarrow$
  f$\in$NON-EMPTY $\Leftrightarrow$ $\exists$<x,t>[STP(f,x,t)] – re upper bound

  EMPTY = L_e = { f | $\forall$ x φ_f (x) $\uparrow$ }
  f$\in$EMPTY $\Leftrightarrow$ $\forall$<x,t>[~STP(f,x,t)] – co-re upper bound

  are the next sets we will study.

L_{ne} is re

- $\mathbf{L_{ne}}$ is enumerated by

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

- This assumes that **0** is in $\mathbf{L_{ne}}$ since **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 $\mathbf{L_{ne}}$.

L_{ne} is Non-Recursive

- Note in the previous enumeration that **F** is a function of just one argument, as we are using an extended pairing function **<x,y,z> = <x,<y,z>>**.
- We can show $\mathbf{Halt = L_u = K_0} \le \mathbf{L_{ne}}$ This shows $\mathbf{L_{ne}}$ cannot be recursive, for if it were then $\mathbf{K_0}$ is recursive.
- Let **f, x** be a pair of natural number where **f** is an arbitrary function index and **x** is an arbitrary input.
- Define $\mathbf{f_x(y) = \varphi_f(x)}$, for all $\mathbf{y} \in \aleph$.
- Then $<\mathbf{f, x}> \in \mathbf{K_0} \Leftrightarrow \forall \mathbf{y f_x(y)}\downarrow \Rightarrow \mathbf{f_x} \in \mathbf{L_{ne}}$
  else $<\mathbf{f, x}> \notin \mathbf{K_0} \Leftrightarrow \forall \mathbf{y f_x(y)}\uparrow \Rightarrow \mathbf{f_x} \notin \mathbf{L_{ne}}$
- Thus, a decision procedure for $\mathbf{L_{ne}}$ (equivalently for $\mathbf{L_e}$) implies one for $\mathbf{K_0}$.

L_{ne} is re by Quantification

- Can do by observing that

  $$\mathbf{f \in L_{ne} \Leftrightarrow \exists <x,t> STP(f, x, t)}$$

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

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

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

Many-One Degrees

- The relationship **A** $\equiv_m$ **B** is an equivalence relationship (why?)
- If **A** $\equiv_m$ **B**, we say **A** and **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**, $\mathbf{A} \le_1 \mathbf{B}$,
  if there exists a total recursive 1-1 function **f**
  such that
  $\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{f(x)} \in \mathbf{B}$
- We say that **A** is one-one equivalent to **B**,
  $\mathbf{A} \equiv_1 \mathbf{B}$, if $\mathbf{A} \le_1 \mathbf{B}$ and $\mathbf{B} \le_1 \mathbf{A}$
- 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 **A** and **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**, **A** $\leq_t$ **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**, **A** $\equiv_t$ **B**, if **A** $\leq_t$ **B** and **B** $\leq_t$ **A**
- Sets that are Turing equivalent are in a very loose sense equally hard or easy.

Turing Degrees

- The relationship **A** $\equiv_t$ **B** is an equivalence relationship (why?)
- If **A** $\equiv_t$ **B**, we say **A** and **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 Halt = K_0 = L_u

- **Halt = K_0 = L_u = { <f, x> | φ_f (x) $\downarrow$}**
- Let **A** be an arbitrary re set. By definition, there exists an effective procedure φ_a, such that **dom(φ_a) = A**. Put equivalently, there exists an index, **a**, such that **A = W_a**.
- **x** $\in$ **A** iff **x** $\in$ **dom(φ_a)** iff φ_a(x)$\downarrow$ iff **<a,x>** $\in$ **K_0**
- The above provides a 1-1 function that reduces **A** to **K_0** (**A** $\leq_1$ **K_0**)
- Thus, the universal set, **Halt = K_0 = L_u**, is an re (1-1, m-1, Turing) complete set.

The Set K

- $\mathbf{K = \{ f \mid \varphi_f(f)\downarrow \}}$
- Define $\mathbf{f_x(y) = \varphi_f(x)}$, for all **y**. The index for $\mathbf{f_x}$ can be computed from **f** and **x** using $\mathbf{S_{1,1}}$, where we add a dummy argument, **y**, to $\mathbf{\varphi_f}$. Let that index be $\mathbf{f_x}$. (Yeah, that's overloading.)
- $\mathbf{\langle f,x \rangle} \in \mathbf{K_0}$ iff $\mathbf{x} \in \mathbf{dom(\varphi_f)}$ iff $\forall \mathbf{y[\varphi_{f_x}(y)\downarrow]} \Rightarrow \mathbf{f_x} \in \mathbf{K}$
  $\mathbf{\langle f,x \rangle} \notin \mathbf{K_0}$ iff $\mathbf{x} \notin \mathbf{dom(\varphi_f)}$ iff $\forall \mathbf{y[\varphi_{f_x}(y)\uparrow]} \Rightarrow \mathbf{f_x} \notin \mathbf{K}$
- The above provides a 1-1 function that reduces $\mathbf{K_0}$ to **K,** i.e., $\mathbf{K_0} \leq_1 \mathbf{K}$.
- Since $\mathbf{K_0}$ is an re (1-1, m-1, Turing) complete set and **K** is re, then **K** is also re (1-1, m-1, Turing) complete.

More re Complete Sets

- HasIdentity (HId) = $\{ f \mid \exists x\ f(x) = x \}$
  $f \in \text{HId} \Leftrightarrow \exists\langle x,t\rangle\ [\ \text{STP}(f,x,t)\ \&\&\ \text{VALUE}(f,x,t)=x\ ]$
- Halt $\leq$ HId;
  Given $\langle f,x\rangle$, define $g_{f,x}(y) = f(x) - f(x) + y$
  $\langle f,x\rangle \in \text{Halt} \Leftrightarrow \forall y[g_{f,x}(y)=y] \Rightarrow g_{f,x} \in \text{HId}$
  $\langle f,x\rangle \notin \text{Halt} \Leftrightarrow \forall y[g_{f,x}(y)\uparrow] \Rightarrow g_{f,x} \notin \text{HId}$
- HasZero (HZ) = $\{ f \mid \exists x\ f(x) = 0 \}$
  $f \in \text{HZ} \Leftrightarrow \exists\langle x,t\rangle\ [\ \text{STP}(f,x,t)\ \&\&\ \text{VALUE}(f,x,t)=0\ ]$
  Given $\langle f,x\rangle$, define $g_{f,x}(y) = f(x) - f(x)$
  $\langle f,x\rangle \in \text{Halt} \Leftrightarrow \forall y[g_{f,x}(y)=0] \Rightarrow g_{f,x} \in \text{HZ}$
  $\langle f,x\rangle \notin \text{Halt} \Leftrightarrow \forall y[g_{f,x}(y)\uparrow] \Rightarrow g_{f,x} \notin \text{HZ}$
- This shows HId and HZ are re-complete

Id Equivalent to TOT

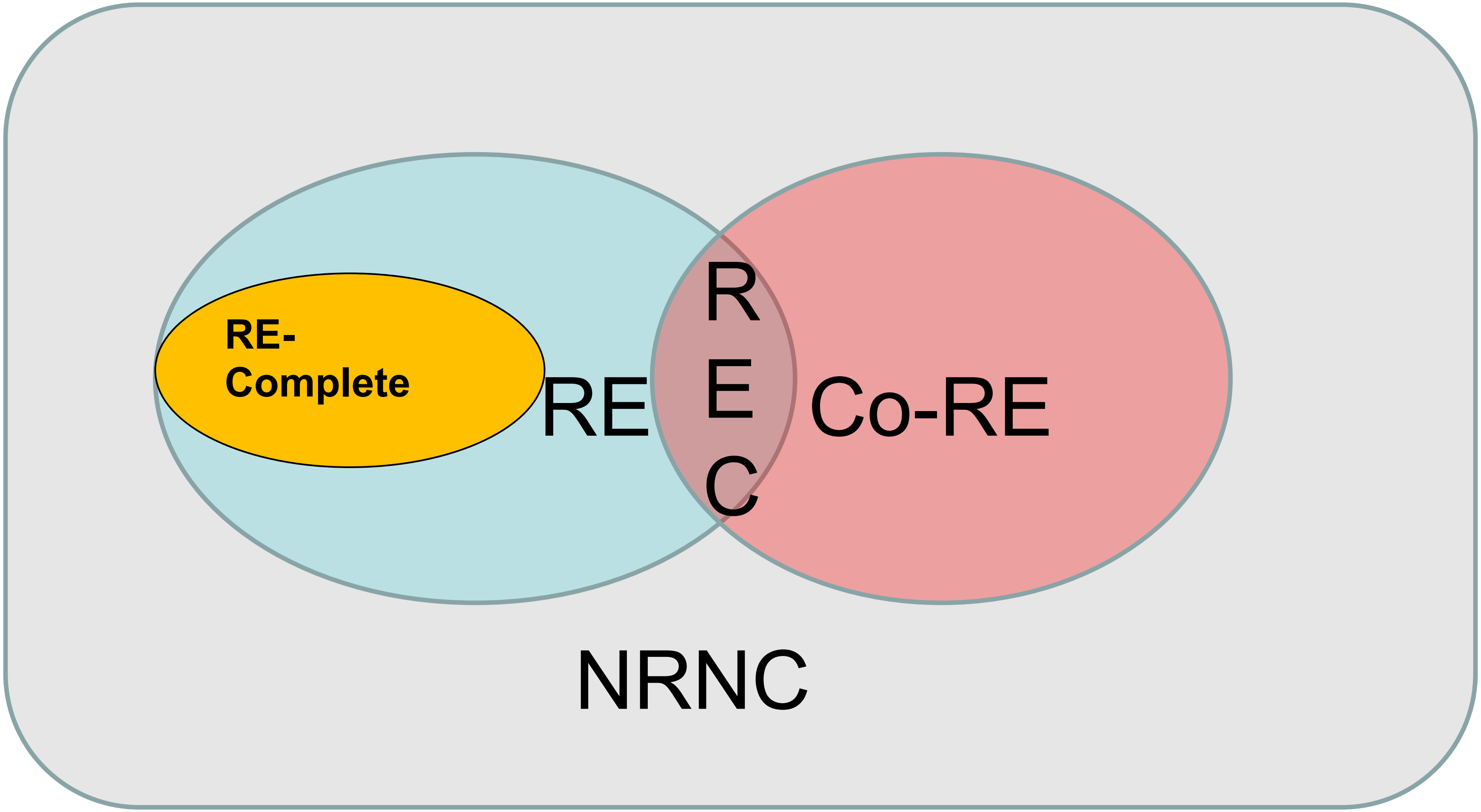
- **Identity (Id) = { f | $\forall$x f(x) = x }**
  f $\in$ id $\Leftrightarrow$ $\forall$x$\exists$t [STP(f,x,t) && VALUE(f,x,t)=x]
- **TOT $\leq$ Id;**
  Given f, define $g_f(x) = f(x) - f(x) + x$
  f $\in$ TOT $\Leftrightarrow$ $\forall$x[$g_f(x)$=x] $\Leftrightarrow$ $g_f \in$ Id
  Id $\leq$ TOT;
  Given f, define $g_f(x) = \mu y\ [f(x) = x]$
  f $\in$ Id $\Leftrightarrow$ $\forall$x[$g_f(x)\downarrow$] $\Leftrightarrow$ $g_f \in$ TOT
- **TOT $\equiv_1$ Id**

MI Equivalent to TOT

- MonoIncreasing (MI) = $\{ f \mid \forall x\ f(x+1) > f(x) \}$
  $f \in MI \Leftrightarrow \forall x \exists t\ [\ STP(f,x,t)\ \&\&\ STP(f,x+1,t)$
  $\&\&\ VALUE(f,x+1,t) > VALUE(f,x,t)\ ]$
- TOT ≤ MI;
  Given f, define $g_f(x) = f(x) - f(x) + x$
  $f \in TOT \Leftrightarrow \forall x[g_f(x)=x] \Rightarrow g_f \in MI$
  $f \notin TOT \Leftrightarrow \exists x[g_f(x)\uparrow] \Rightarrow g_f \notin MI$
- MI ≤ TOT;
  Given f, define $g_f(x) = \mu y\ [f(x+1) > f(x)]$
  $f \in MI \Leftrightarrow \forall x[g_f(x)\downarrow] \Leftrightarrow g_f \in TOT$
- $MI \equiv_1 TOT$

The Overall Picture

UNIVERSE OF SETS



NonRE = (NRNC ∪ Co-RE) - REC

Reduction and Rice’s

What Rice's is All About

- Let **P** be some property of **re languages.**
- Let $\mathbf{L_P}$ be the set of languages having property **P**.
- For example, if **P** is the property that a language is infinite then $\mathbf{L_P} = \{ \mathbf{L} \mid \mathbf{L}$ is infinite re $\}$.
- Said differently, $\mathbf{L_P} = \{ \mathbf{f} \mid \mathbf{domain}(\varphi_\mathbf{f})$ is infinite $\}$ or even $\mathbf{L_P} = \{ \mathbf{g} \mid \mathbf{range}(\varphi_\mathbf{g})$ is infinite $\}$.
- In addition to limiting its applicability to properties of re languages (alternatively procedures that define those languages), Rice's also limits its focus to properties that are **non-trivial**. A property is **trivial** if it applies to **all re sets** (procedures) or **no re sets** (procedures).

Rice Focused on Procedures

- As noted, we can view re languages by talking about the procedures that define them (either by domain or range or even both).
- Rice’s Theorem deals with properties that split the set of procedures (really their indices) in two partitions: those procedures that have the property and those that do not have the property.
- For Rice’s Theorem to apply, the property must be about the non-implementation dependent behaviors of procedures, not how they might be implemented or even how efficient they are.

Operational Aspects of Rice

- Operationally, this means if **f** and **g** are two arbitrary function indices such that $\forall \mathbf{x}\ \varphi_{\mathbf{f}}(\mathbf{x}) = \varphi_{\mathbf{g}}(\mathbf{x})$ then Rice's theorem cannot differentiate **f** from **g**. That is, either **f** and **g** both have the property or neither does.

- This means, if we want to differentiate procedures based on how fast they compute values, Rice's Theorem is useless.

- The above is what I call the Strong Version of Rice's Theorem

Weaker Versions of Rice

- We will show that if **f** and **g** are two arbitrary function indices such that **domain** φ_f **= domain** φ_g then Rice's theorem cannot differentiate **f** from **g**.
- We will also show that if **f** and **g** are two arbitrary function indices such that **range** φ_f **= range** φ_g then Rice's theorem cannot differentiate **f** from **g**.
- The domain form is what Rice originally proved but we extend it here to three versions so one can choose the one that is best for a particular property.

Non-Trivial versus Trivial

- An example probably bring this home best
- Let **Doppel =**
  { f | there is a **g≠f** where **∀x** $\boldsymbol{\varphi_f(x) = \varphi_g(x)}$ **}**
- It is the case that all procedures have twins (doppelgangers) so **Doppel** applies to all procedures and **Doppel** is the set of Natural Numbers.
- For this reason, Rice's does not apply.

Comment about Ø

- Consider the set of indices of procedures that have empty domains.
- If **f** and **g** have the same domain then **domain(f) = Ø** ⇔ **domain(g) = Ø**.
- By our previous discussion Rice's Theorem cannot differentiate **f** from **g** as it cannot look at how each computes its domain. It can only observe that both have the same domain (**Ø**), so they are indistinguishable.
- Moreover, if **f** and **g** have empty domains, they have empty ranges and they diverge everywhere, so they cannot be distinguished by their ranges or their mappings from input to output.

Rice's Original Theorem

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

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

is undecidable. Note again that membership in $\mathbf{L_P}$ is based purely on the domain of a function, not on any aspect of its implementation.

Note: Rice's Theorem focuses on re sets as being domains of procedures. As noted previously, we will extend this to also consider ranges of procedures and even functional properties (mappings from domains to ranges.)

Rice's Proof-1

Proof: We will assume, *wlog*, that **P** does not contain **Ø** (functions that have empty domains). If it does, we switch our attention to the complement of **P**. Now, since **P** is non-trivial, there exists some language **L** with property **P**. Let **[r]** be a procedure (recursive function) whose domain is **L** (**r** is the index of a semi-decision procedure for **L**). Suppose **P** were decidable. We will use this decision procedure and the existence of **r** to decide **Halt** (**K_0**).

Rice's Proof-2

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

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

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

If the latter were true, it might be able to differentiate φ_r from $\mathbf{F}_{x,y.r}$. as they have different indices (different implementations) even if they might have the same domains.

I/O Property

- An I/O property, $\mathscr{P}$, 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, **f** and **g**, are such that $\varphi_{\mathbf{f}}$ and $\varphi_{\mathbf{g}}$ converge on the same inputs and, when they converge, produce precisely the same result, then both **f** and **g** must have property $\mathscr{P}$, 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 or procedures are I/O properties.

Example of Non-I/O Property

- **X10 = { f | ∀x f(x)** converges by **10** units of time }
- Can describe as
  f ∈ X10 ⇔ ∀x [STP(f,x,10)]
- Consider two TMs to compute the function **Z(x) = 0**
- Can do by TM **Z1** with one instruction, **R**. **Z1 ∈ X10**
- Can do by less efficient TM **Z2**, $\mathcal{L}\,\mathcal{R}$**R**. **Z2 ∉ X10**
- We now have two implementations of **Z,** one of which has the **X10** property and the other of which does not have the **X10** property, yet both have the same I/O behavior. Thus, **X10** is not an I/O property and Rice's Theorem has nothing to say about the decidability or undecidability of membership in **X10**.
- I will tell you, however, that **X10** is co-re, unsolvable.

Strong Rice's Theorem

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

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

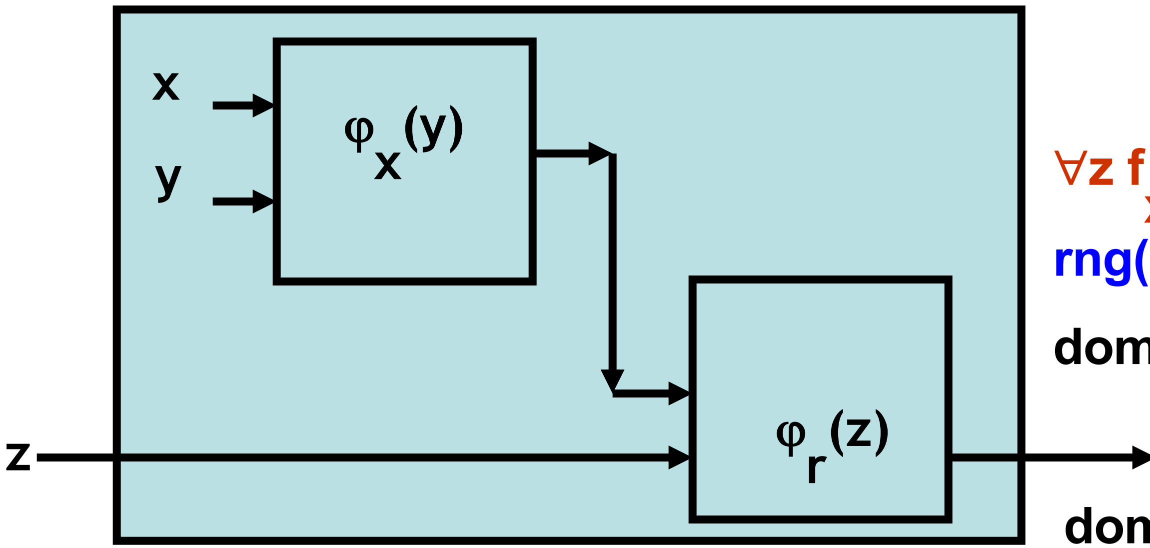
is undecidable.

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

Strong Rice's Proof

- Given **x**, **y**, **r**, where **r** is in the set $\mathbf{S}_{\mathscr{P}}\mathbf{.= \{f \mid} \varphi_{\mathbf{f}}$ has property $\mathscr{P}$},
  define the function
  $\mathbf{f_{x,y,r}(z) = \varphi_x(y) - \varphi_x(y) + \varphi_r(z)}$
- $\mathbf{f_{x,y,r}(z) = \varphi_r(z)}$ if $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow$ **;**
  Otherwise $\forall\mathbf{z}\ \mathbf{f_{x,y,r}\ (z)}\uparrow$ and $\mathbf{dom(f_{x,y,r}\ ) = \emptyset}$.
  Thus, $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow$ iff $\mathbf{f_{x,y,r}}$ has property $\mathscr{P}$, and so
  $\mathbf{K_0} \leq \mathbf{S}_{\mathscr{P}}$**.**

Picture Proof



$\forall z\ f_{x,y,r}(z)=\varphi_r(z)$ If $\varphi_x(y)\downarrow$

$rng(f_{x,y,r})=rng(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=dom(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$rng(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$\exists z\ f_{x,y,r}(z)\neq\varphi_r(z)$ If $\varphi_x(y)\uparrow$

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 $\mathscr{P}$ be some non-trivial, implementation-independent property of the indices of recursive functions. Then

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

is undecidable.

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

Weak Rice's Theorem2: Let $\mathscr{P}$ be some non-trivial, implementation-independent property of the indices of recursive functions. Then

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

is undecidable.

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

Corollaries to Rice's

Corollary: The following properties of re sets are undecidable

a) **L** = **Ø**

b) **L** is finite

c) **L** is a regular set

d) **L** is a context-free set

For all of these, Rice's shows that the complements are undecidable but operationally it doesn't matter.

Both the property and its complement are non-trivial, and each is implementation-independent.

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 = $\{ f \mid$ for some $x, y, x \neq y, f(x)\downarrow$ and $f(y)\downarrow$ 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 = $\{ f \mid$ for all $x, y, f(x)\downarrow$ and $f(y)\downarrow$, 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 ≠ y, f(x)↓ and f(y)↓ and f(x) ≠ f(y) }**
   b) **NoRepeats = { f | for all x, y, f(x)↓ and f(y)↓, and x ≠ y ⇒ f(x) ≠ 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** ∈ **A** and **y** ∈ **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 **φ_f(x)** converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = φ_f(x)**.

Sample Questions#2,3

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

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

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

Sample Question#4

4. Let **P = { f | ∃ x [STP(f, x, x)] }**. Why does Rice's theorem not tell us anything about the undecidability of **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

   E = { z | z = x + y, where x $\in$ S and y $\in$ 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 | ∃K [M** halts in at most **K** steps independent of its starting configuration **] }**
- **CTime** cannot be shown undecidable by Rice's Theorem as it breaks property 2 (it is implementation dependent)
  - Choose **M1** and **M2** to each Standard Turing Compute (STC) **ZERO**
  - **M1** is **R** (move right to end on a zero)
  - **M2** is $\mathcal{L}$ $\mathcal{R}$ **R** (time is dependent on argument length)
  - **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 [STP(M, C, K)] }**
- This would appear to imply that **CTime** is not even re. However, a TM that only runs for **K** steps can scan at most **K** distinct tape symbols. Thus, if we use unary notation, **CTime** can be expressed
- **CTime = { M | $\exists$K $\forall$$C_{|C|\le K}$ [STP(M, C, 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 (maybe)

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) ~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 **φ_f(x)** converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = φ_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

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

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

Sample Question#8

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

Sample Question#9

9. 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**?

Sample Question#10

10. Define the pairing function **$<x,y>$** and its two inverses **$<z>_1$** and **$<z>_2$**, where if **$z = <x,y>$**, then **$x = <z>_1$** and **$y = <z>_2$**.

Sample Question#11

11. Assume **A** $\leq_{\mathbf{m}}$ **B** and **B** $\leq_{\mathbf{m}}$ **C**.
    Prove **A** $\leq_{\mathbf{m}}$ **C**.

Sample Question#12

12. Let **P = { f | ∃ x [STP(f, x, x)] }**. Why does Rice's theorem not tell us anything about the undecidability of **P**?

Worked Out Samples

$D2 = \{ f \mid | Domain(f) | > 1 \}$

Use Rice's Theorem to prove that **D2** is undecidable. Be Complete.
Note **f∈D2 ⇔ ∃<x,y,t> [STP(f,x,t) & STP(f,y,t) & x≠y]**

D2 is non-trivial as **C0(x) = 0** ∈ **D2** and ↑ ∉ **D2**

Let f,g be two arbitrary indices of procedures such that Domain(f) = Domain(g)

f ∈ **D2** ⇔ | **Domain(f) | > 1**

⇔ | **Domain(g) | > 1**

as **Domain(f) = Domain(g)**

and so cardinalities are the same

⇔ **g** ∈ **D2**

Thus, **D2** is undecidable by Rice's Weak Form#1

R2= { f | | Range(f) | > 1 }

Use Rice's Theorem to prove that **R2** is undecidable. Be Complete.
Note **f∈R2 ⇔ ∃<x,y,t> [STP(f,x,t) & STP(f,y,t) & // skipped x≠y?**
(VALUE(f,x,t) ≠ VALUE(f,y,t))]

R2 is non-trivial as **S(x) = x+1** ∈ **R2** and **C0(x) = 0** ∉ **R2**

Let f,g be two arbitrary indices of procedures such that
Range(f) = Range(g)

f ∈ **R2** ⇔ | **Range(f) | > 1**

⇔ | **Range(g) | > 1**

as **Range(f) = Range(g)**

and so cardinalities are the same

⇔ **g** ∈ **R2**

Thus, **R2** is undecidable by Rice's Weak Form#2

HasDup(HD)={ f | $\exists$x,y, x≠y, f(x)↓, f(y)↓, & f(x)=f(y) }

Show a minimal quantification of some known primitive recursive predicate that provides an upper bound for the complexity of **HD**.

$\exists$<x,y,t> [STP(f,x,t) & STP(f,y,t) & x≠y & (VALUE(f,x,t) = VALUE(f,y,t))]

Thus, **HD $\leq_m$ K_0**

HasDup(HD)={ f | $\exists$x,y, x$\neq$y, f(x)$\downarrow$, f(y)$\downarrow$, & f(x)=f(y) }

Use Rice's Theorem to prove that **HD** is undecidable. Be Complete.

HD is non-trivial as **C0(x) = 0** $\in$ **HD** and **S(x) = x+1** $\notin$ **HD**

Let **f,g** be two arbitrary indices of procedures such that

$\forall$**x f(x) = g(x)**

f $\in$ **HD** $\Leftrightarrow \exists$**x,y [(x$\neq$y), f(x)$\downarrow$, f(y)$\downarrow$ & f(x) = f(y))**

$\Leftrightarrow \exists$**x,y [(x$\neq$y), g(x)$\downarrow$, g(y)$\downarrow$ & g(x) = g(y))**

as $\forall$**x f(x) = g(x)** and so **g** has same I/O properties

$\Leftrightarrow$ **g** $\in$ **HD**

f $\notin$ **HD** $\Leftrightarrow \forall$**x,y [(x$\neq$y)** $\Rightarrow$ either **f(x)**$\uparrow$ or **f(y)**$\uparrow$ or **f(x) $\neq$ f(y))**

$\Leftrightarrow \forall$**x,y [(x$\neq$y)** $\Rightarrow$ either **g(x)**$\uparrow$ or **g(y)**$\uparrow$ or **g(x) $\neq$ g(y))**

as $\forall$**x f(x) = g(x)** and so **g** has same I/O properties

$\Leftrightarrow$ **g** $\notin$ **HD**

Thus, **HD** is undecidable by Rice's Strong Form

HasDup(HD)={ f | ∃x,y, x≠y, f(x)↓, f(y)↓, & f(x)=f(y) }

Show that **K = { f | f(f)** converges **}** is many-one reducible to **HD**.

Let **f** be an arbitrary index.

From **f**, define $\forall \mathbf{x}\ \mathbf{F_f(x) = f(f)-f(f)}$.

f ∈ **K** implies $\forall \mathbf{x}\ \mathbf{F_f(x) = 0}$ implies $\mathbf{F_f} \in \mathbf{HD}$.

f ∉ **K** implies $\forall \mathbf{x}\ \mathbf{F_f(x)}\uparrow$ implies $\mathbf{F_f} \notin \mathbf{HD}$.

Thus, $\mathbf{K \le_m HD}$

HasDup(HD)={ f | $\exists$x,y, x$\neq$y, f(x)$\downarrow$, f(y)$\downarrow$, & f(x)=f(y) }

Show that **HD** is many-one reducible to
K = { f | f(f) converges **}**

Let **f** be an arbitrary index. From **f**, define
$\forall \mathbf{z}\ \mathbf{F_f(z) = \exists<x,y,t> [STP(f,x,t) \ \&\ STP(f,y,t) \ \&\ x\neq y \ \&\ (VALUE(f,x,t) = VALUE(f,xyt))]}$
$\mathbf{f} \in \mathbf{HD} \Rightarrow \forall \mathbf{z}\ \mathbf{F_f(z)}\downarrow \Rightarrow \mathbf{F_f(F_f)}$ converges $\Rightarrow \mathbf{F_f} \in \mathbf{K}$
$\mathbf{f} \notin \mathbf{HD} \Rightarrow \forall \mathbf{z}\ \mathbf{F_f(z)}\uparrow \Rightarrow \mathbf{F_f} \notin \mathbf{K}$.

Thus, $\mathbf{HD \leq_m K}$

AllDup(AD) = { f | ∀x f(x)↓ & ∃y, x ≠ y, where f(y)↓ & f(x) = f(y) }

Show a minimal quantification of some known primitive recursive predicate that provides an upper bound for the complexity of **AD**.

∀x ∃<y,t> [STP(f,x,t) & STP(f,y,t) & x≠y
& **(VALUE(f,x,t) = VALUE(f,y,t))]}**

AD looks to be up there with **TOT**

AllDup(AD) = { f | $\forall x\ f(x)\downarrow$ & $\exists y, x \neq y$, where $f(y)\downarrow$ & $f(x) = f(y)$ }

Use Rice's Theorem to prove that **AD** is undecidable. Be Complete.

AD is non-trivial as **$C0(x) = 0$** $\in$ **AD** and **$S(x) = x+1$** $\notin$ **AD**

Let f,g be two arbitrary indices of procedures such that $\forall x\ f(x) = g(x)$

f $\in$ **AD** $\Leftrightarrow$ $\forall$**x** $f(x)\downarrow$ & $\exists$**y**, $x\neq y$, where **$f(y)\downarrow$ & $f(x) = f(y)$**

$\Leftrightarrow$ given any **x**, $f(x)\downarrow$ & $\exists$**y**$\neq$x, **$f(y)\downarrow$ & $f(x) = f(y)$**

$\Leftrightarrow$ given any **x**, $g(x)\downarrow$ & $\exists$**y**$\neq$x, **$g(y)\downarrow$ & $g(x) = g(y)$**

as $\forall$**x $f(x) = g(x)$** and so **g** has same I/O properties

$\Leftrightarrow$ **g** $\in$ **AD**

Thus, **HD** is undecidable by Rice's Strong Form

AllDup(AD) = { f | ∀x f(x)↓ & ∃y, x ≠ y, where f(y)↓ & f(x) = f(y) }

Show that **TOT = { f |** for all **x**, f**(x)** converges **}** is many-one reducible to **AD**.

Let **f** be an arbitrary index.

From **f**, define ∀**x** $\mathbf{F_f(x) = f(x)\text{-}f(x)}$.

f ∈ **TOT** ⇒ ∀**x** $\mathbf{F_f(x) = 0}$ ⇒ $\mathbf{F_f}$ ∈ **AD**.

f ∉ **TOT** ⇒ ∃**x** $\mathbf{F_f(x)}\uparrow$ ⇒ $\mathbf{F_f}$ ∉ **AD**.

Thus, $\mathbf{TOT \le_m AD}$

AllDup(AD) = { f | $\forall$x f(x)$\downarrow$ & $\exists$y, x ≠ y, where f(y)$\downarrow$ & f(x) = f(y) }

Show that **AD** is many-one reducible to

TOT = { f | for all **x**, f**(x)** converges **}**

Let **f** be an arbitrary index.

From **f**, define $\forall$**x** $\mathbf{F_f(x)} = \exists\mathbf{y}$ **[**x≠y & **f(x) = f(y)]**

$\mathbf{f} \in \mathbf{AD} \Rightarrow \forall\mathbf{x}\ \mathbf{F_f(x)}\downarrow \Rightarrow \mathbf{F_f} \in \mathbf{TOT}$

$\mathbf{f} \notin \mathbf{AD} \Rightarrow \exists\mathbf{x}\ \mathbf{F_f(x)}\uparrow \Rightarrow \mathbf{F_f} \notin \mathbf{TOT}$.

Thus, $\mathbf{AD} \leq_m \mathbf{TOT}$

Note it is rare to us**e** $\exists$ without using **STP** but I am fine with the search failing as a result of **f** diverging on either **x** or some **y** or failing in a never-ending search.

Challenge

Semi-Constant(SC) = { f | ∃C, ∀x f(x)↓ ⇒ f(x) = C }

Note: ↑ ∈ **SC** and **C_0(x)=0** ∈ **SC**

Can describe as **f** ∈ **SC** ⇔

∃C ∀$\langle x,t \rangle$ [STP(f,x,t) ⇒ VALUE(f,x,t) = C]

This implies **SC** is as hard as **Non-TOT={ f |∃x f(x)**↑ **}** as

f ∈ **Non-TOT** ⇔ **∃x ∀t [~STP(f,x,t)]**

However, **SC** only takes one quantifier and is undecidable (one of the weaker versions of Rice shows its undecidability).

I can tell you that either **SC** $\equiv_m$ **HALT** or **SC** $\equiv_m$ **Non-HALT** where **Non-HALT = { $\langle f,x \rangle$ | f(x)**↑ **}**.

Your job is to figure out which and rewrite the quantifier expression. You should also apply Rice's to verify undecidability.

Rewriting Systems

Post Systems

Thue Systems

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

Semi-Thue Systems

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

Word Problems

- Let **S = (Σ, 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.

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 α**qa**β is represented by the string hα**qa**βh, **a** being the scanned symbol.
- The tuple **q a b s** leads to
  qa → sb
- Moving right and left can be harder due to blanks.

Details of Halt(TM) $\leq$ 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 $\mathbf{h1^xq_10h}$, seeking to accept **x** (enter $\mathbf{q_0}$) or reject (run forever).
- Add rules
  - $\mathbf{q_0a \rightarrow q_0}$ $\quad\quad\quad\quad\quad\quad$ $\forall \mathbf{a} \in \mathbf{\{0,1\}}$
  - $\mathbf{bq_0 \rightarrow q_0}$ $\quad\quad\quad\quad\quad\quad$ $\forall \mathbf{b} \in \mathbf{\{0,1\}}$

- The added rule allows us to "erase" the tape if we accept **x**.
- This means that acceptance can be changed to generating $\mathbf{hq_0h}$.

- The next slide shows the consequences.

Semi-Thue Word Problem

- Construction from TM, **M**, gets:
- $\mathbf{h1^xq_10h \Rightarrow_{\Sigma(M)}{}^* hq_0h}$ **iff** $\mathbf{x \in \mathcal{L}(M)}$**.**
- $\mathbf{hq_0h \Rightarrow_{\Pi(M)}{}^* h1^xq_10h}$ **iff** $\mathbf{x \in \mathcal{L}(M)}$**.**
- $\mathbf{hq_0h \Leftrightarrow_{\Sigma(M)}{}^* h1^xq_10h}$ **iff** $\mathbf{x \in \mathcal{L}(M)}$**.**
- 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, CFG and CSG) produce recursive sets.
- Type 0 grammars are as powerful at generating (producing) re sets as Turing machines are at enumerating them (Constructive Proof later).

Formal Language

Undecidability Continued

PCP and Traces

Post Correspondence Problem

- Many problems related to grammars can be shown no harder than PCP
- Each instance of PCP is denoted by an integer n>0, a finite alphabet Σ and two n-tuples of non-λ words over Σ
  $x = (x_1, \ldots, xn), y = (y_1, \ldots, y_n)$
- The decision problem is to determine if there is a sequence of indices $i_1, \ldots, i_k$, $1 \leq i_j \leq n$, for $1 \leq j \leq k$, where indices can be repeated, such that
  $x_{i1} \ldots x_{ik} = y_{i1} \ldots y_{ik}$

PCP Example#1

- Let n=3, Σ = {a,b}
  x = (a b a, b b, a); y = (b a b, b, b a a)
- Must start with the second elements as they both start with b and all others misalign.
- A solution is 2, 3, 1, 2
  b b a a b a b b = b b a a b a b b
- Note 2,3,1,2,2,3,1,2 is also a solution

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	$\underline{\text{ab}}$	aa	$\underline{\text{aa}}$	a	$\underline{\text{a}}$	]
[	$\underline{\text{aba}}$	aba	$\underline{\text{a}}$	a	$\underline{\text{b}}$	b	$\underline{*}$aa]
  - And

*	$\underline{*}$	a	$\underline{\text{a}}$	b	$\underline{\text{b}}$
$\underline{*}$	*	$\underline{\text{a}}$	a	$\underline{\text{b}}$	b

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 $\underline{*}W_f]$
- 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

  $\mathbf{S \rightarrow A \mid B}$

  $\mathbf{A \rightarrow x_i\, A\, [i] \mid x_i\, [i]} \qquad \mathbf{1 \le i \le n}$

  $\mathbf{B \rightarrow y_i\, B\, [i] \mid y_i\, [i]} \qquad \mathbf{1 \le i \le n}$

  $\mathbf{A \Rightarrow^* x_{i_1} \ldots x_{i_k} [i_k] \ldots [i_1]} \qquad \mathbf{k > 0}$

  $\mathbf{B \Rightarrow^* y_{i_1} \ldots y_{i_k} [i_k] \ldots [i_1]} \qquad \mathbf{k > 0}$

- 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.
- As intersection of two CFLs is a CSL, this is one proof we can determine is a CSG produces no (or any or finite or infinite) strings.

CSG Produces Something

S	$\rightarrow x_i\ S\ y_i^R \mid x_i\ T\ y_i^R \quad 1 \le i \le n$
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 **Ø** otherwise.
- Thus, **P** has a solution iff
  - **L(G) ≠ Ø**
  - **L(G) is infinite**
- Alternate proof to one on intersections of CFLs (there are many alternate approaches to this one)

Traces and Grammars

Traces

- A valid trace
  - $\# C_1 \# C_2 \# C_3 \# C_4 \ \ldots \# C_{k-1} \# C_k \#$, 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
  - $\# C_1 \# C_2 \# C_3 \# C_4 \ \ldots \# C_{k-1} \# C_k \#$, 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 **M**, is a word of the form
$\# C_0 \# C_1 \# C_2 \# C_3 \# \ldots \# C_{k-1} \# C_k \#$

  where $\mathbf{C_i \Rightarrow C_{i+1}\ 0 \le i < k, C_0}$ is a starting configuration and $\mathbf{C_k}$ is a terminating configuration.
- We allow some laxness, where the configurations might be encoded in a manner appropriate to the machine model.

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,
  $\{ \# C_1 \# C_2 \# C_3 \# C_4 \ \ldots \# C_{k-1} \# C_k \# \mid$
  $k \geq 1$ and $C_i \Rightarrow_M C_{i+1}$, 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
  $\{ \# C \# C \# C \# C \ \ldots \# C \# C \# \}$ as it never changes the tape contents or its state. It's as hard as $\{ww \mid w \in \{a,b\}^*\}$

Using Pumping Lemma

- From previous slide, assume that the language of traces,
  L = { # C # C # C # C … # 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| ≤ 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| ≤ 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 for differences in only **three** characters in 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.

Details of Traces as CSL

- Easiest starting point is not Turing Machines but rather FRS's with Residue
- Rules are of form
  $ax + b \rightarrow cx + d$
  a, b, c, d are natural numbers,
  $1 \leq b < a$; $1 \leq d < c$
- Can show that these systems do not require order as do FRS's
- Residues can check for non-divisibility

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

  j. $\quad INC_r[i] \qquad p_{n+j}\ x \rightarrow p_{n+i}\ p_r\ x$
  j. $\quad DEC_r[s, f] \qquad p_{n+j}\ p_r\ x \rightarrow p_{n+s}\ x$
  $\qquad p_{n+j}\ p_r\, x + k\, p_{n+j} \rightarrow p_{n+f}\ p_r\, x + k\, p_{n+f}\ ,\ 1 \le k < p_r$

  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

  $$0 \rightarrow 0$$

Non-Traces as Existential

- 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.

Non-Traces as Just One Error

- 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 for an error here is is similar to what was described with the LBA and a single step check and can be done with a stack.
- What the stack cannot do is look at sequences longer than single pairs.

Intersection of CFLs

- Assume computations starts on ID $\mathbf{X_0}$ and ends on a unique ID $\mathbf{Z_0}$.
- Language of even odd pairs is **L1**,

  $\mathbf{L1 = \{ \#Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \# \}}$

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

  This checks the even/odd steps of an even length computation.
- Language of of even pairs is **L2**,

  $\mathbf{L2 = \{ \#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k}\# Z_0 \# \}}$

  where $\mathbf{X_{2i-1} \Rightarrow X_{2i} , 1 \le i \le k.}$

  This checks the odd/even steps of an even length computation.
- Both **L1** and **L2** can be shown to be CFLs

Intersection Continued

- Now, $\mathbf{X_0}$ is chosen as some selected input ID to a machine, and $\mathbf{Z_0}$ is the unique ID on which the machine halts. But,

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

  where $\mathbf{X_i \Rightarrow X_{i+1}}$, $\mathbf{0 \le i < 2k}$, and $\mathbf{X_{2k} \Rightarrow Z_0}$.
- This checks <u>all</u> steps of an even length computation. But our original system halts if and only if it produces $\mathbf{Z_0}$ in an even (also odd) number of steps.
- Thus, the intersection is non-empty just in case the machine eventually halts when started on $\mathbf{X_0}$.
- This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.

Consequence of Partial Traces Being CFLs

- 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 **L1** and **L2**, where **L1** $\cap$ **L2 ≠ Ø** , 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.

Intersection of CFLs (precise)

- Let (n, ((a1,b1,c1,d1) , … ,(ak,bk,ck,dk)) be some factor replacement system with residues. Define grammars G1 and G2 by using the 4k+2 rules

  G : $F_i \rightarrow 1^{ai}F_i1^{ci} \mid 1^{ai+bi}\#1^{ci+di} \quad 1 \le i \le k$

  $S_1 \rightarrow \# F_i S_1 \mid \# F_i \# \quad 1 \le i \le k$

  $S_2 \rightarrow \# 1^{x0}S_11^{z0}\#$ $\quad Z_0$ is 0 for us

  G1 starts with S_1 and G2 with S_2
- Thus, using the notation of writing Y in place of 1^Y,

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

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

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

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

  where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$.

  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,

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

where $X_i \Rightarrow X_{i+1}$, $0 \le i < 2k$, and $X_{2k} \Rightarrow Z_0$. This checks <u>all</u> 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 some machine **M** (details are model dependent)
  - 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

What's Undecidable?

- We cannot decide if the set of valid terminating traces of an arbitrary machine **M** is non-empty (machine accepts something) – $\boldsymbol{L}$**(M) ≠ Ø**
- 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.

$L = \Sigma^*?$

- 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

Quotients of CFLs (concept)

Let L1 = L(G1) = { $ # Y_0 # Y_1 # Y_2 # Y_3 # … # Y_{2j} # Y_{2j+1} # }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
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 # … # X_{2k-1} # X_{2k}# Z_0 #}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ 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 | 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, ((a1,b1,c1,d1) , … ,(ak,bk,ck,dk)) be some factor replacement system with residues. Define grammars G1 and G2 by using the 4k+4 rules

  G : F_i → $1^{ai}F_i1^{ci}$ | $1^{ai+bi}\#1^{ci+di}$ $1 \le i \le k$
  T_1 → $\# F_i T_1$ | $\# F_i \#$ $1 \le i \le k$
  A → 1 A 1 | \$ #
  S_1 → $\$T_1$
  S_2 → $A\ T_1 \# 1^{z0} \#$ Z_0 is 0 for us

  G1 starts with S_1 and G2 with S_2
- Thus, using the notation of writing Y in place of 1^Y,

  L1 = L(G1) = { \$ #Y_0 # Y_1 # Y_2 # Y_3 # … # Y_{2j} # Y_{2j+1} # }
  where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
  This checks the even/odd steps of an even length computation.
  But, L2 = L(G2) = { X \$ #X_0 # X_1 # X_2 # X_3 # X_4 # … # X_{2k-1} # X_{2k}# Z_0 # }
  where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ 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 \$.

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 | the system F 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 C0. Our rules will be

S	→	**# C0 #**	**where C0 = αq0aβ is initial ID**
q a	→	**s b**	**if q a b s ∈ T**
b q a x	→	**b a s x**	**if q a R s ∈ T, a,b,x ∈ Σ**
b q a #	→	**b a s B #**	**if q a R s ∈ T, a,b ∈ Σ**
# q a x	→	**# a s x**	**if q a R s ∈ T, a,x ∈ Σ, a≠B**
# q a #	→	**# a s B #**	**if q a R s ∈ T, a ∈ Σ, a≠B**
# q a x	→	**# s x #**	**if q a R s ∈ T, x ∈ Σ, a=B**
# q a #	→	**# s B #**	**if q a R s ∈ T, a=B**
b q a x	→	**s b a x**	**if q a L s ∈ T, a,b,x ∈ Σ**
# q a x	→	**# s B a x**	**if q a L s ∈ T, a,x ∈ Σ**
b q a #	→	**s b a #**	**if q a L s ∈ T, a,b ∈ Σ, a≠B**
# q a #	→	**# s B a #**	**if q a L s ∈ T, a ∈ Σ, a≠B**
b q a #	→	**s b #**	**if q a L s ∈ T, b ∈ Σ, a=B**
# q a #	→	**# s B #**	**if q a L s ∈ 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 | w $\in$ L(G) and m≥0 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≥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 another proof of this one result.
- For Type 0, emptiness and even the membership problems are undecidable.

Undecidability

- **Is L =$\varnothing$, for CSL, L?** **PCP reduction**
- **Is L=Σ*, 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 $L_1 \cap L_2 = \varnothing$ 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 ~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
- Finiteness of 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: $S \rightarrow h1^x q_1 0h$

  $hq_0h \rightarrow V$

  $V \rightarrow aV$

  $V \rightarrow \lambda$

- $x \in \mathcal{L}(M)$ 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

Traces: Representing IDs

- We create a trace of a Turing Machine model, each ID is of the form α**qa**β where α is the shortest string representing all non-blanks to the left of the scanned square and β is the shortest string representing all non-blanks to the left of the scanned square. **q** is the state and **a** is the character at the scanned square.
- For a Turing Machine model using binary tapes
  r = (1(0+1)* + λ) q (0+1) (λ + (0+1)*1)
  where **q = (q1 + q2 + …qn)** if **Q= {q1,q2, … , qn}**
- A trace-like string is then either
  t = (#r)$^+$#
- As trace-like is Regular so are strings that don't look like traces since Regular are closed un complement

Traces: Single Steps

- A single step of any derivation in **M** is a pair of configurations $\mathbf{C_1\#C_2\#}$ where $\mathbf{C_1 \Rightarrow_M C_2}$.
- For Turing machines, because the strings are over multiple letters, a pair like $\mathbf{C_1\#C_2\#}$ is as hard to generate as words of the form
{ **ww** | **w** is over a multi-letter alphabet }.
- To circumvent this, we represent TM pairs by reversing one of the two IDs, e.g., $\mathbf{C_1\#C_2^R\#}$ or $\mathbf{C_1^R\#C_2\#}$

Why CFLs?

- I have glossed over some complexity
- A single step of a TM can be checked by a Deterministic PDA. As DCFLS are closed under complement, we have that incorrect single steps are also DCFLs
- Showing the single step checker is not hard but ahgain requires that we are looking at either $\mathbf{C_1\#C_2^R\#}$ or $\mathbf{C_1^R\#C_2\#}$

Traces: Multiple Steps

- A multistep derivation in **M** is a sequence of configurations $\mathbf{C_1\#C_2\#\ \ldots\ \#C_k\#}$ where $\mathbf{C_i \Rightarrow_M C_{i+1}}$.
- Multistep (>1) traces are Context Sensitive.
- For Turing machines, we often alternate between unreversed and reversed as in $\mathbf{C_1\#C_2^R\#C_3\#C_4^R\ \ldots}$
- This reversing is not needed by the CSG but is convenient when we want to consider that a CFL can get every other pair right or can get a bad trace by just messing up once.

Constant Time Again

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

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 $\Sigma^* \subseteq L^* = \cup_{n\geq 0} L^n \subseteq L$**
  - **first inclusion follows from (1); second from (2)**

Finite Power Problem for Turing Machine Model

- **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.**
- **$L_1 = \{ C_1\# C_2^R \$ \mid C_1, C_2$ are configurations $\}$**
- **$L_2 = \{ C_1\#C_2^R\$C_3\#C_4^R \ \dots \ \$C_{2k-1}\#C_{2k}^R\$ \mid$ where $k \geq 1$ and, for some i, $1 \leq i < 2k$, $C_i \Rightarrow_M C_{i+1}$ is false $\}$**
- **$L = L_1 \cup L_2 \cup \{\lambda\}$**
- **L is context free.**

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

- Any product of L_1 and L_2, which contains L_2 at least once, is L_2.
  For instance, $L_1 \bullet L_2 = L_2 \bullet L_1 = L_2 \bullet L_2 = L_2$.
- 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.

Sketch that CT ≡ Uniform Halting for Infinite Markings

- If **M** is in **CT** then it halts in a fixed time, then input read is independent of tape markings
- This implies **M** halts even if infinitely marked.
- If **M** halts under all markings, even infinite ones, then there must be some bound on its running time that is independent of its input.
- This implies **M** is in **CT**.

Revisiting PCP

Constrained to a single letter

Single Letter Alphabet PCP

- $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_n)$, $\Sigma = \{a\}$
- Can recast as just lengths
- If $\exists i, 1 \le i \le n, |x_i| = |y_i|$, then say YES
- If $\forall i, 1 \le i \le n, |x_i| > |y_i|$, then say NO
- If $\forall i, 1 \le i \le n, |x_i| < |y_i|$, then say NO
- Else $\exists i,j, i \ne j, 1 \le i,j \le n, |x_i| > |y_i|$ and $|x_j| < |y_j|$
  Let $s = |x_i| - |y_i|$ and $t = |y_j| - |x_j|$
  Solution i repeated t times, j repeated s times
  x part is $|x_i| * (|y_j| - |x_j|) + |x_j| * (|x_i| - |y_i|)$
  y part is $|y_i| * (|y_j| - |x_j|) + |y_j| * (|x_i| - |y_i|)$
  $|x_i|*|y_j| - |x_i|*|x_j| + |x_j|*|x_i| - |x_j|*|y_i| = |x_i|*|y_j| - |x_j|*|y_i|$
  $|y_i|*|y_j| - |y_i|*|x_j| + |y_j|*|x_i| - |y_j|*|y_i| = |x_i|*|y_j| - |x_j|*|y_i|$

More 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

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

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

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

c) ~A: can be non-re. If ~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 ~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 | ∃<x,t> [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 | z = x + y, where x $\in$ S 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) = $\exists$<x,y,t> [stp(f_S, x, t) && stp(f_T, y, t) && (z = value(f_S, x, t) + 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 = $\aleph$, then E = { x | 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 | $\forall$x φ_f(x+1) > φ_f(x) }

Let f be arb.

Define G_f(x) = φ_f(x) - φ_f(x) + x

f $\in$ TOTAL iff $\forall$xφ_f(x)$\downarrow$ iff $\forall$x G_f(x)$\downarrow$ iff

$\forall$x φ_f(x) - φ_f(x) + x = x implies G_f $\in$ Incr

f $\notin$ TOTAL iff $\exists$xφ_f(x)$\uparrow$ iff $\exists$x G_f(x)$\downarrow$ iff

$\exists$x (φ_f(x) - φ_f(x) + x)$\downarrow$ implies G_f $\notin$ Incr

You can overload **f** to be index and function, i.e., **f(x)** as φ_f**(x)**

Sample Question

Let **Incr = { f | ∀x, φ_f(x+1)>φ_f(x) }**.
Let **TOTAL = { f | ∀x, φ_f(x)↓ }**.
Prove that **Incr $\leq_m$ TOTAL**.

Let f be arb.

Define G_f (x) = ∃t[stp(f,x,t) && stp(f,x+1,t) && (value(f,x+1,t) > value(f,x,t))]

f ∈ Incr iff ∀x φ_f(x+1)>φ_f(x) iff ∀x G_f (x)↓ iff G_f ∈ TOT

Sample Question

Let **Incr = { f | $\forall$x φ_f(x+1)>φ_f(x) }**.
Use Rice's theorem to show **Incr** is not recursive.

Non-Trivial as
$C_0(x)=0 \notin$ Incr; S(x)=x+1 $\in$ Incr
Let f,g be arb. Such that $\forall x\ \varphi_f(x)=\varphi_g(x)$
f $\in$ Incr iff $\forall x\ \varphi_f(x+1)>\varphi_f(x)$ iff
$\forall x\ \varphi_g(x+1)>\varphi_g(x)$ iff g $\in$ 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 = $\aleph$, then T could be any subset of $\aleph$. There are an uncountable number of such subsets and some are clearly in each of the categories above.

Sample Question

Let **P = { f | ∃ x [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.**

True/False

- Membership in Context-Sensitive Languages is unsolvable
- Membership in Regular Languages can be solved in linear time
- We cannot decide if the intersection of two CFLs is a CFL
- The Post Correspondence Problem over a one-letter alphabet is undecidable

Multiple Choice

- We learned that quantification can be used to determine the upper bound of the complexity of some problem. The general form of such expressions is **Q [Algorithmic Predicate }**. Here **Q** is a sequence of alternating quantifiers. Usually, the predicate is Primitive Recursive involving the functions **STP** and **VALUE**. I will give you a set (membership in it is the problem) and an associated predicate. The Quantifier part won't be specified, and you will need to choose from a set of options what that quantifier is.
- S = { f | Domain(f) is infinite }
- $f \in S \Leftrightarrow Q$ [STP(f,x,t) & x>y }
- Q is the part you must pick from a list

Matching

- When do variants of Rice's apply. For example, which version would you use for $S = \{ f \mid \text{Range}(f) \text{ is infinite } \}$?
- What is an inherent property of some problem, e.g., HALT is re-complete, TOTAL is non-re

Closures of Languages

- A Regular and B CFL. What about $A \cap B$?
  - Can it be Regular
  - Can it be a CFL, non-Regular
  - Can it be a CSL, non-CFL
- A Recursive non-empty, B non-re. What about $A \cap B$?
  - Can it be Recursive
  - Can it be RE non-Recursive
  - Can it be non-RE

Others

- Properties of various language classes and procedures for analyzing language classes.