University of Central Florida

Discrete II Theory of Computation

Charles E. Hughes

Supplemental

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 are 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 $\mathbf{id' \notin R_M}$

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 $\underline{0}$ 0 0 …
- The underscore indicates the square being read. We denote this by the finite id
  1 0 1 0 0 1 1 q7 $\underline{0}$
- In this notation, we always write down the scanned square, even if it and all symbols to its right are blank.

More on Encoding of TM

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

Simulation by RM

```
1.         DEC3[2,q0]            : Go to simulate actions in state 0
2.         DEC3[3,q1]            : Go to simulate actions in state 1
…
n.         DEC3[ERR,qn-1]        : Go to simulate actions in state n-1
…
qj.        IF_r1_ODD[qj+2]       : Jump if scanning a 1
qj+1.      JUMP[set_k]           : If (qj 0 0 qk) is rule in TM
qj+1.      INC1[set_k]           : If (qj 0 1 qk) is rule in TM
qj+1.      DIV_r1_BY_2           : If (qj 0 R qk) is rule in TM
           MUL_r2__BY_2
           JUMP[set_k]
qj+1.      MUL_r1_BY_2           : If (qj 0 L qk) is rule in TM
           IF_r2_ODD then INC1
           DIV_r2__BY_2[set_k]
…
set_n-1.   INC3[set_n-2]         : Set r3 to index n-1 for simulating state n-1
set_n-2.   INC3[set_n-3]         : Set r3 to index n-2 for simulating state n-2
…
set_0.     JUMP[1]               : Set r3 to index 0 for simulating state 0
```

Fixups

- Need epilog so action for missing quad (halting) jumps beyond end of simulation to clean things up, placing result in **r1**.
- Can also have a prolog that starts with arguments in first n registers 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 r1, 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.   INC2[4]         : putting answer in r2
4.   DIV_r1_BY_2[2]  : strip a 1 from r1
5.   DEC1[5,6]       : Set r1 to 0 (prepare for answer)
6.   DEC2[6,7]       : Copy r2 to r1
7.   INC1[6]         :
8.                   : Answer is now in r1
```

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**-register machine, having m instructions.

  Encode the contents of registers **r1,…,rn** by the powers of $\mathbf{p_1,\ldots p_n}$.

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

  Use **pn+m+1** as prime factor that indicates simulation is done.
- This is in essence the 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]

$$\mathbf{p_{n+j}x \rightarrow p_{n+i}p_rx}$$

j. DECr[s, f]

$$\mathbf{p_{n+j}p_rx \rightarrow p_{n+s}x}$$

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

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

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

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.

Example of Order

Consider the simple machine to compute **r1:=r2 – r3** (limited)

1. **DEC3[2,3]**
2. **DEC2[1,1]**
3. **DEC2[4,5]**
4. **INC1[3]**
5.

Subtraction Encoding

Start with 3^x5^y7

7 • 5 x	→	11 x
7 x	→	13 x
11 • 3 x	→	7 x
11 x	→	7 x
13 • 3 x	→	17 x
13 x	→	19 x
17 x	→	13 • 2 x
19 x	→	x

Analysis of Problem

- If we don't obey the ordering here, we could take an input like $\mathbf{3^5 5^2 7}$ and immediately apply the second rule (the one that mimics a failed decrement).
- We then have $\mathbf{3^5 5^2 13}$, signifying that we will mimic instruction number **3**, never having subtracted the **2** from **5**.
- Now, we mimic copying **r2** to **r1** and get $\mathbf{2^5 5^2 19}$.
- 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

  RULE(F, x) = μ z (1 ≤ z ≤ exp(F, 0)+1) [exp(F, 2*z-1) | x]
- Note: if **x** is divisible by **a_i**, and **i** is the least integer for which this is true, then **exp(F,2*i-1) = a_i** where **a_i** is the number of prime factors of **F** involving **p_{2i-1}**. Thus, **RULE(F,x) = i**.

  If x is not divisible by any **a_i**, **1≤i≤n**, then **x** is divisible by **1**, and **RULE(F,x)** returns **n+1**. That's why we added **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

  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 only if we stop

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.

FRS Subtraction

- $2^0 3^a 5^b \Rightarrow 2^{a-b}$
  $3*5x \to x$ or 1/15
  $5x \to x$ or 1/5
  $3x \to 2x$ 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 ≤ z ≤ 4) [exp(F, 2*z-1) | x]
  RULE (F,$3^4\ 5^2$) = 1, as 15 divides $3^4\ 5^2$
- NEXT(F, x) = (x // exp(F, 2*RULE(F, x)-1)) * exp(F, 2*RULE(F, x))
  NEXT(F,$3^4\ 5^2$) = ($3^4\ 5^2$ // 15 * 1) = $3^3 5^1$
  NEXT(F,$3^3\ 5^1$) = ($3^3\ 5^1$ // 15 * 1) = 3^2
  NEXT(F,3^2) = (3^2 // 3 * 2) = $2^1 3^1$
  NEXT(F, $2^1 3^1$) = ($2^1 3^1$ // 3 * 2) = 2^2
  NEXT(F, 2^2) = (2^2 // 1 * 1) = 2^2

Rest of simulation

- 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

  $\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} \underline{0} \ldots$

  and ends with

  $\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} 0 1^{y} \underline{0} \ldots$

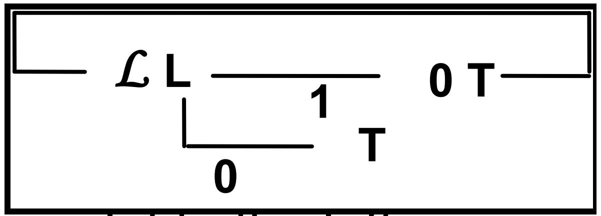
  where **y = F(x1, … , xn)**.

More Helpers

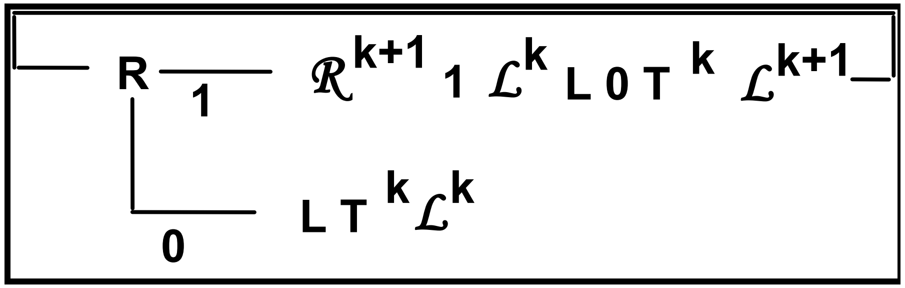
- To build our simulation we need to construct some useful submachines, in addition to the $\mathcal{R}$, $\mathcal{L}$, **R**, **L**, and $\mathbf{C}_k$ machines already defined.
- **T** -- translate moves a value left one tape square
  …$\underline{?}$01^{x}0… ⇒ …?1$^x\underline{0}$0…

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

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



- **Rot$_k$** -- Rotate a **k** value sequence one slot to the left
  …$\underline{0}$1^{x1}01^{x2}0…01^{xk}0…
  ⇒ …$\underline{0}$1^{x2}0…01^{xk}01^{x1}0…



R
1
$\mathcal{R}^{k+1}$ 1 $\mathcal{L}^k$ L 0 T ^{k} $\mathcal{L}^{k+1}$
0
L T ^{k} $\mathcal{L}^k$

Basic Functions

All Basis Recursive Functions are Turing computable:

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

$(R1)^aR$

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

C_{n-i+1}

- $S(x) = x+1$

$C_1 1R$

Closure Under Composition

If **G, H$_1$, … , 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 $\mathbf{E(x_1,\ldots,x_n)}$ 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

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

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 the 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:

$$\begin{tikzcd}
G\mathcal{L}^{n+1} & L \arrow[r, "0"] \arrow[dr, "1"'] & \mathcal{R}^{n+2} \\
& & 0\mathcal{R}^{n+2}H\ \text{Shift}\ \mathcal{L}^{n+2}1 \arrow[ul, bend left]
\end{tikzcd}$$

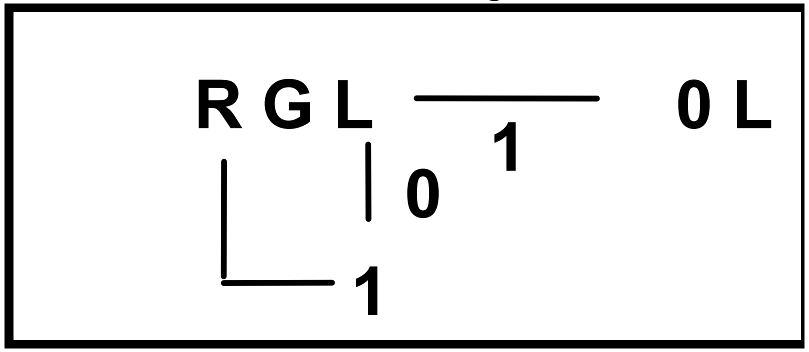
Since our Turing Machine simulator can produce the same value for any arbitrary PRF, F, we show that Turing Machines are closed under induction (primitive recursion).

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

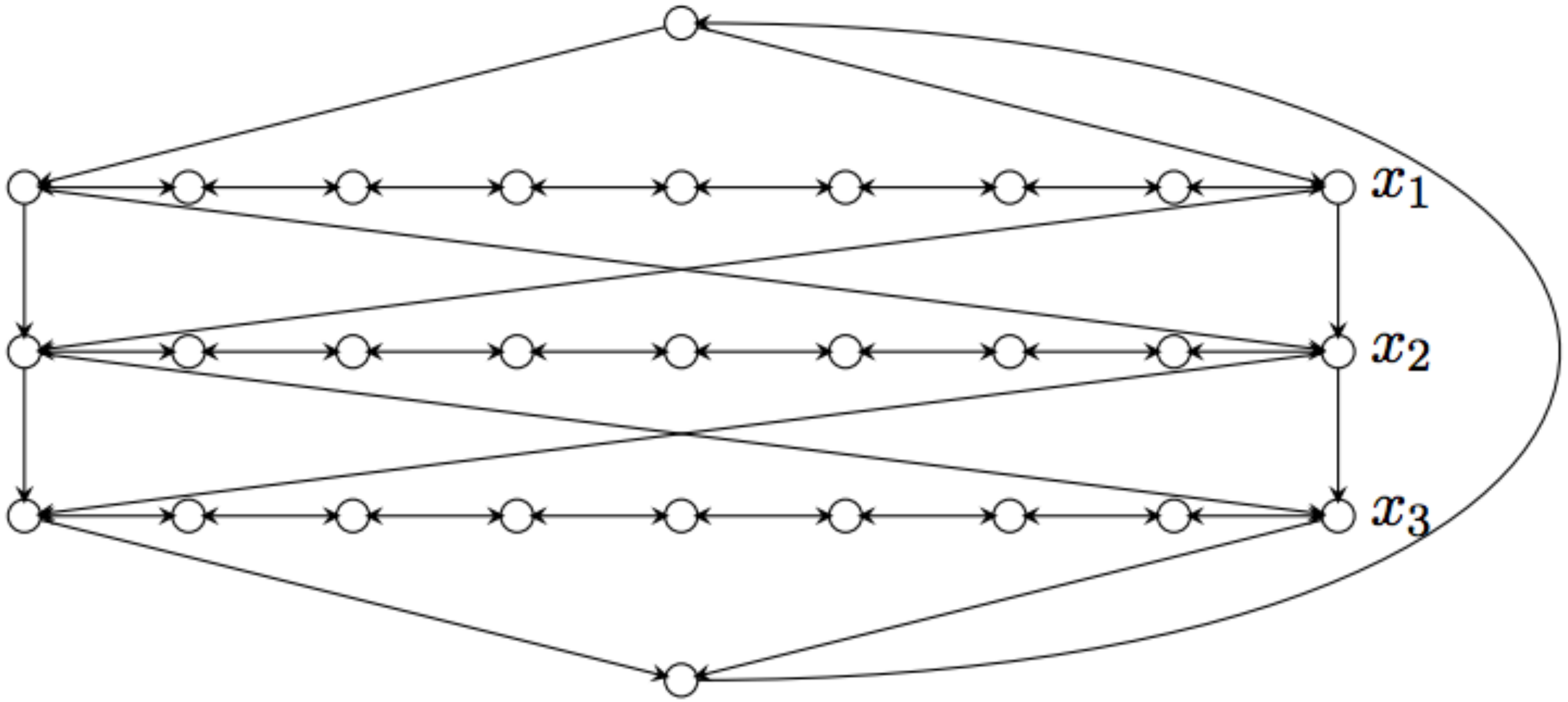


Consequences of Equivalence

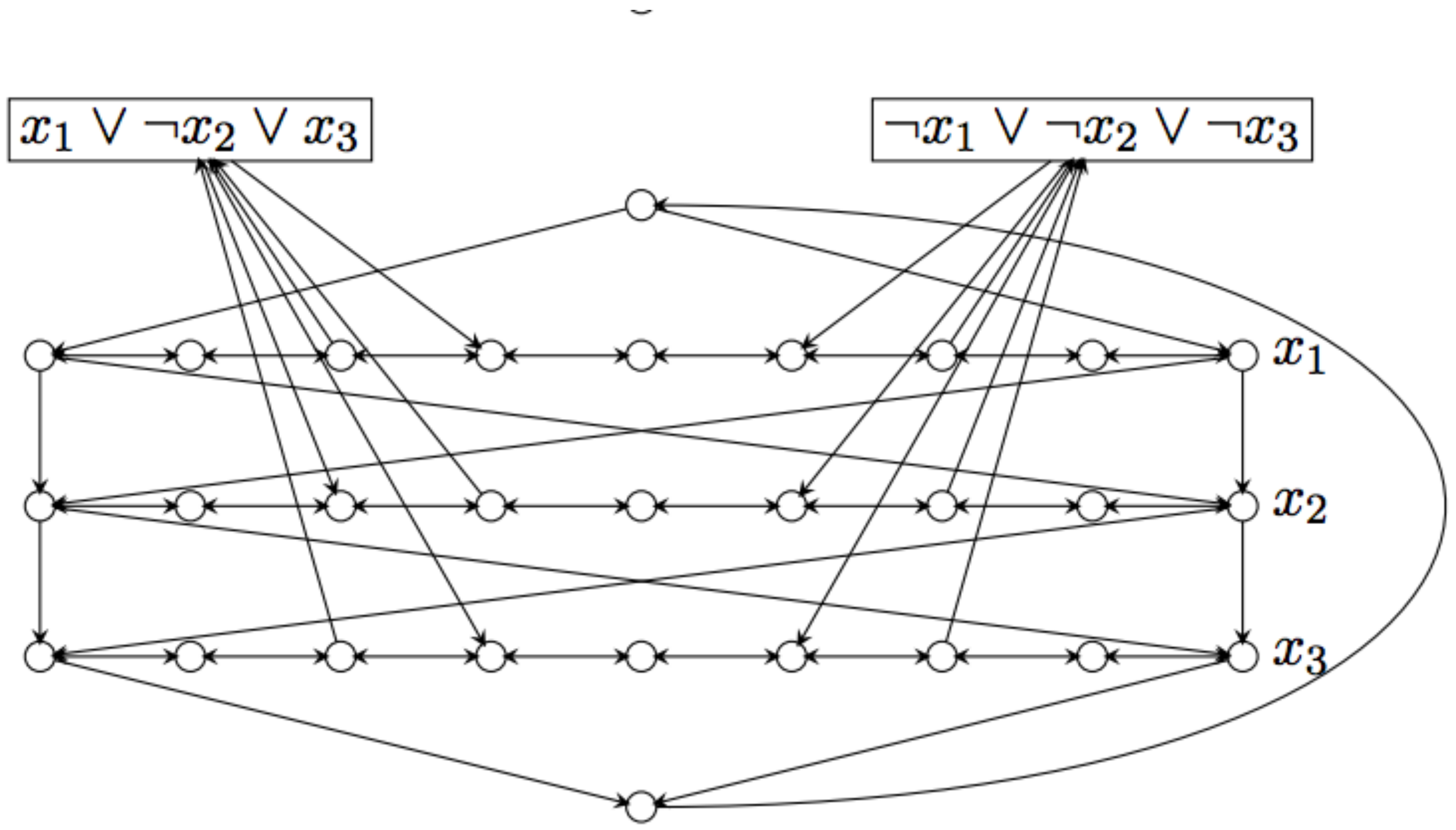
- Theorem: The computational power of Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.
- Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.
- Theorem: Universal machines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

HAMILTONIAN CIRCUIT (HC) DECISION PROBLEM IS NP-HARD

HC Variable Gadget



HC Gadgets Combined


$x_1 \vee \neg x_2 \vee x_3$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x_2
x_3

Hamiltonian Path

- Note we can split an arbitrary node, v, into two (v’,v” – one, v’, has in-edges of v, other, v”, has out-edges. Path (not cycle) must start at v” and end at v’ and goal is still K.

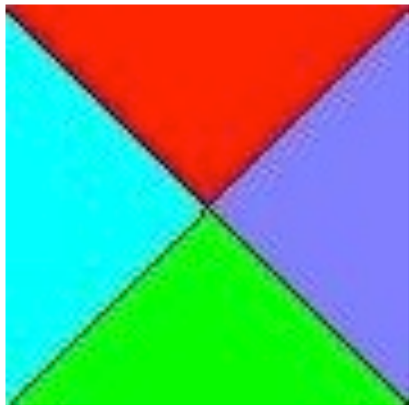
Travelling Salesman

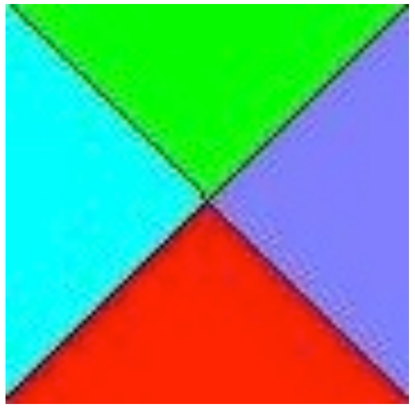
- Start with HC = (V,E), K=|V|
- Set edges from HC instance to 1
- Add edges between pairs that lack such edges and make those weights 2 (often people make these K+1); this means that the reverse of unidirectional links also get weight 2
- Goal weight is K for cycle

Tiling

Undecidable and NP-Complete Variants

Basic Idea of Tiling



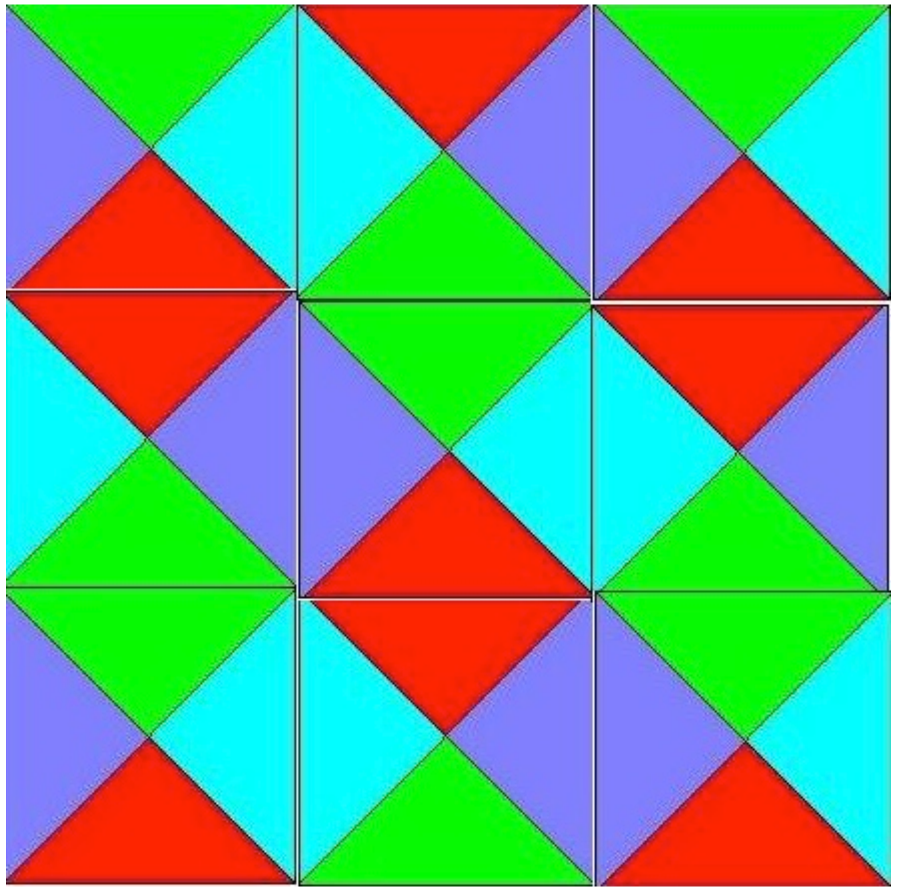


A single tile has colors on all four sides. Tiles are often called dominoes as assembling them follows the rules of placing dominoes. That is, the color (or number) of a side must match that of its adjacent tile, e.g., tile, t2, to right of a tile, t1, must have same color on Its left as is on the right side of t1. This constraint applies to top and as well as sides. Boundary tiles do not have constraints on their sides that touch the boundaries.

Instance of Tiling Problem

- A finite set of tile types (a type is determined by the colors of its edges)
- Some 2d area (finite or infinite) on which the tiles are to be laid out
- An optional starting set of tiles in fixed positions
- The goal of tiling the plane following the adjacency constraints and whatever constraints are indicated by the starting configuration.

A Valid 3 by 3 Tiling of Tile Types from Previous Slide



Some Variations

- Infinite 2d plane (impossible in general)
  - Our two tile types can easily tile the 2d plane
- Finite 2d plane (hard in general)
  - Our two tile types can easily tile any finite 2d plane
  - This is called the Bounded Tiling Problem.

- One dimensional space (hmm?)
- Infinite 3d space (not even semi-decidable in general)

Tiling the Plane

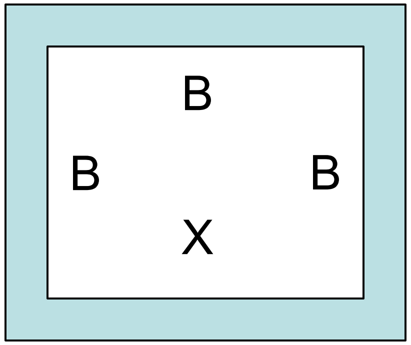
- We will start with a Post Machine, M = (Q, Σ, δ, q_0), with tape alphabet $\Sigma = \{B,1\}$ where B is blank and δ maps pairs from $Q \times \Sigma$ to $Q \times (\Sigma \cup \{R,L\})$. M starts in state q_0
  - (Turing Machine with each action being L, R or Print)
- We will consider the case of M starting with a blank tape
- We will constrain our machine to never go to the left of its starting position (semi unbounded tape)
- We will mimic the computation steps of M
- Termination occurs if in state q reading b and $\delta(q,b)$ is not defined
- We will use the fact that halting when starting at the left end of a semi unbounded tape in its initial state with a blank tape is undecidable

The Tiling Decision Problem

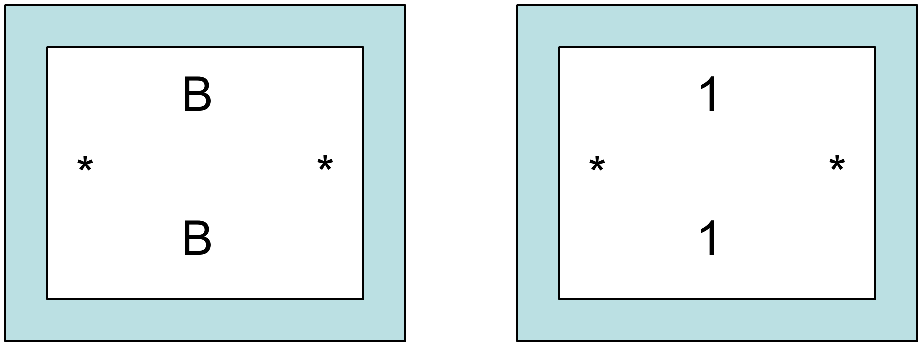
- Given a finite set of tile types and a starting tile in lower left corner of 2d plane, can we tile all places in the plane?
- A place is defined by its coordinates (x,y), x≥0, y≥0
- The fixed starting tile is at (0,0)

Colors

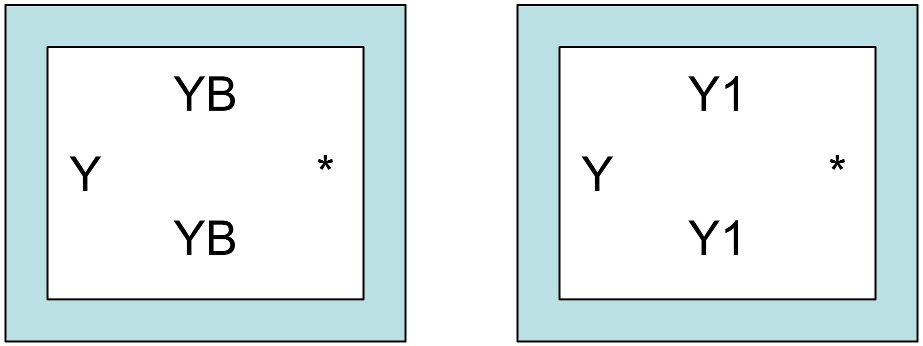
- Given M, define our tile colors as
- $\{X, Y, *, B, 1, YB, Y1\} \cup Q \times \{B,1\} \cup Q \times \{YB,Y1\} \cup Q \times \{R,L\}$
- Simplest tile (represents Blank on X axis)



Tiles for Copying Tape Cell

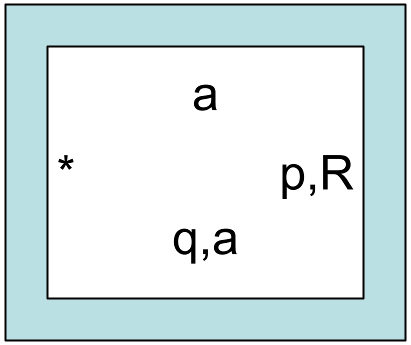


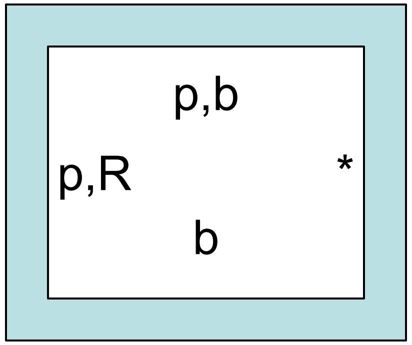
Copy cells not on left boundary and not scanned



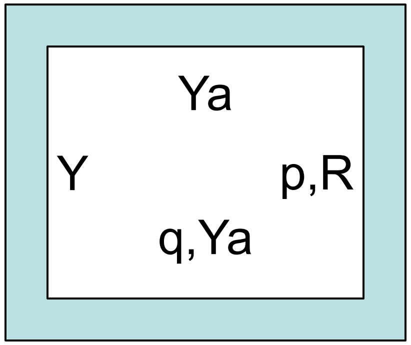
Copy cells on left boundary but not scanned

Right Move δ(q,a) = (p,R)

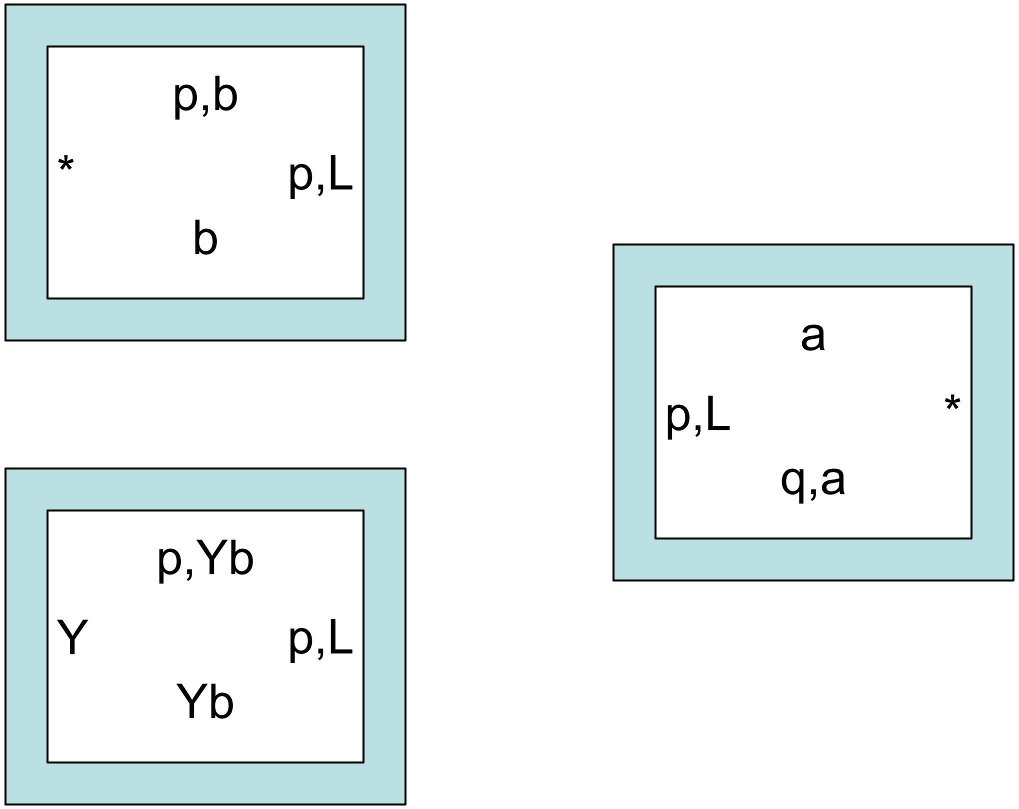




where $b \in \Sigma$

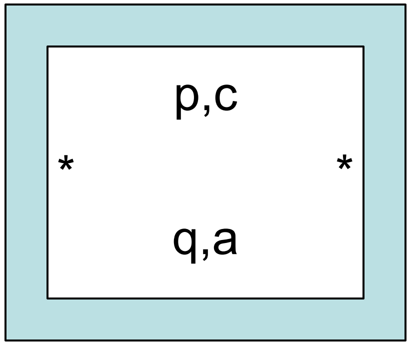


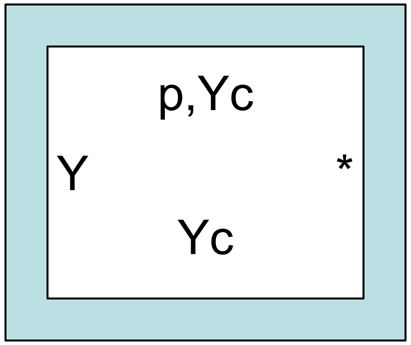
Left Move δ(q,a) = (p,L)



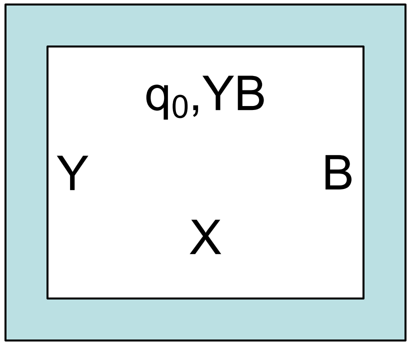
where b∈Σ

Print δ(q,a) = (p,c)

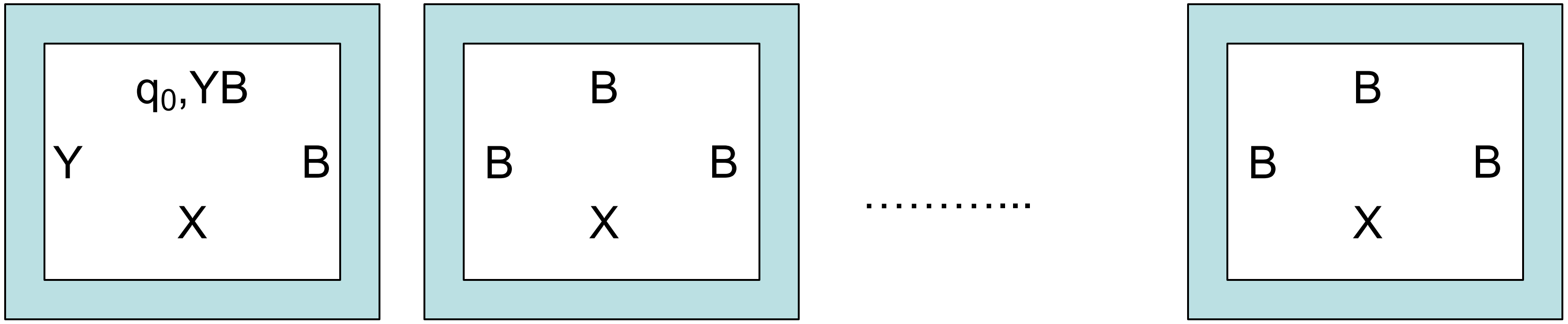




Corner Tile and Bottom Row

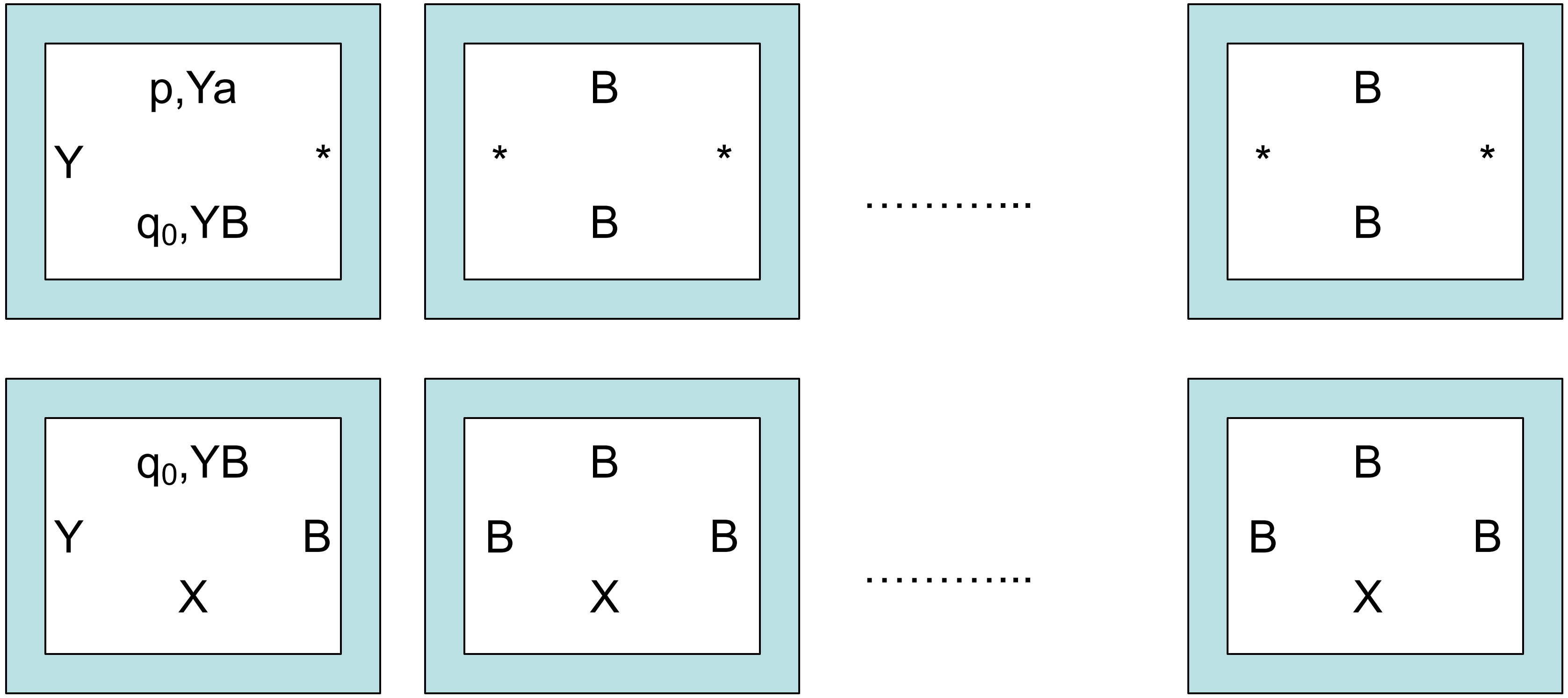


Zero-ed Row is forced to be



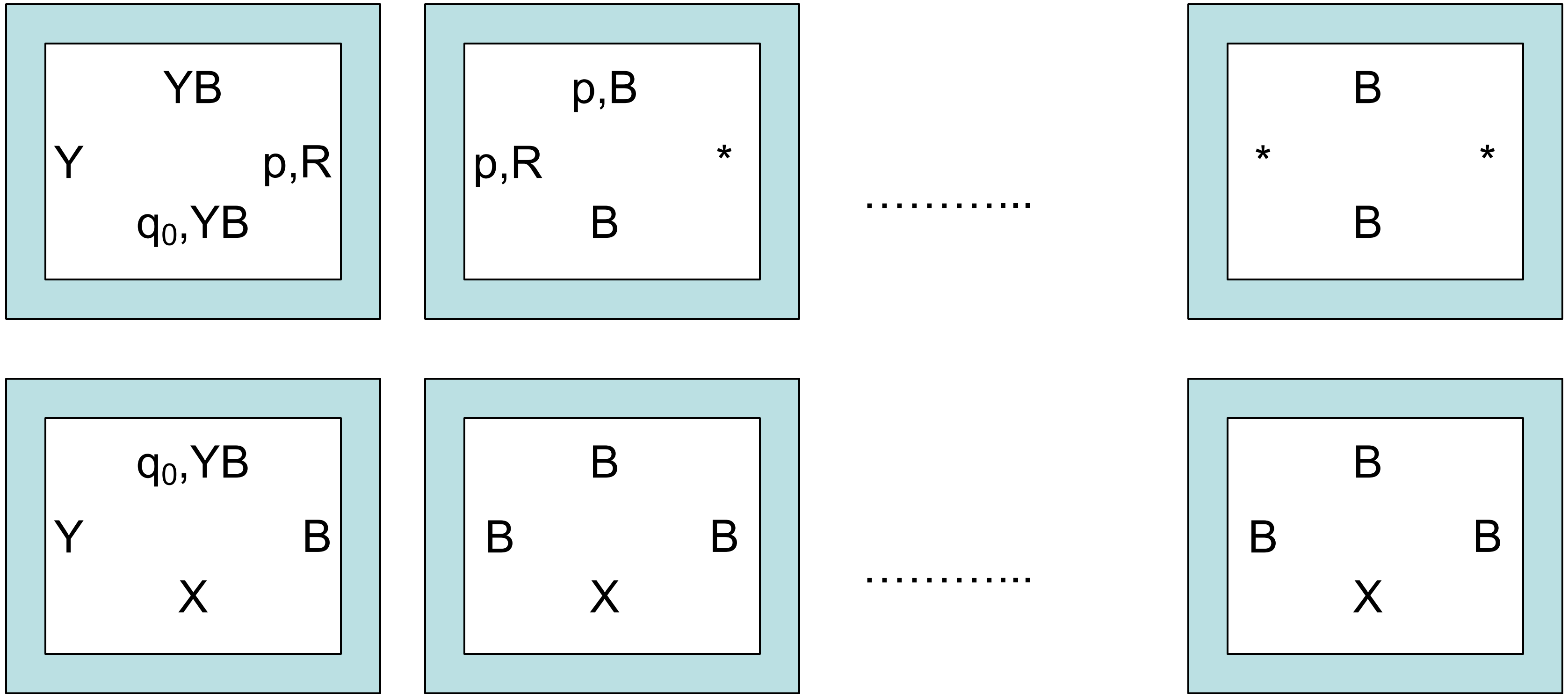
First Action Print

As we cannot move left of leftmost character first action is either right or print. Assume for now that $\delta(q_0,B) = (p,a)$



First Action Right Move

As we cannot move left of leftmost character first action is either right or print.
Assume for now that $\delta(q_0,B) = (p,R)$



The Rest of the Story Part 1

- Inductively we can show that, if the i-th row represents an infinite transcription of the Turing configuration after step i then the (i+1)-st represents such a transcription after step i+1. Since we have shown the base case, we have a successful simulation.

The Rest of the Story Part 2

- Consider the case where M eventually halts when started on a blank tape in state q_0. In this case we will reach a point where no actions fill the slots above the one representing the current state. That means that we cannot tile the plane.
- If M never halts, then we can tile the plane (in the limit).

The Rest of the Story Part 3

- The consequences of Parts 1 and 2 are that Tiling the plane is as hard as the complement of the Halting problem which is co-RE Complete.
- This is not surprising as this problem involves a universal quantification over all coordinates (x,y) in the plane.

Constraints on M

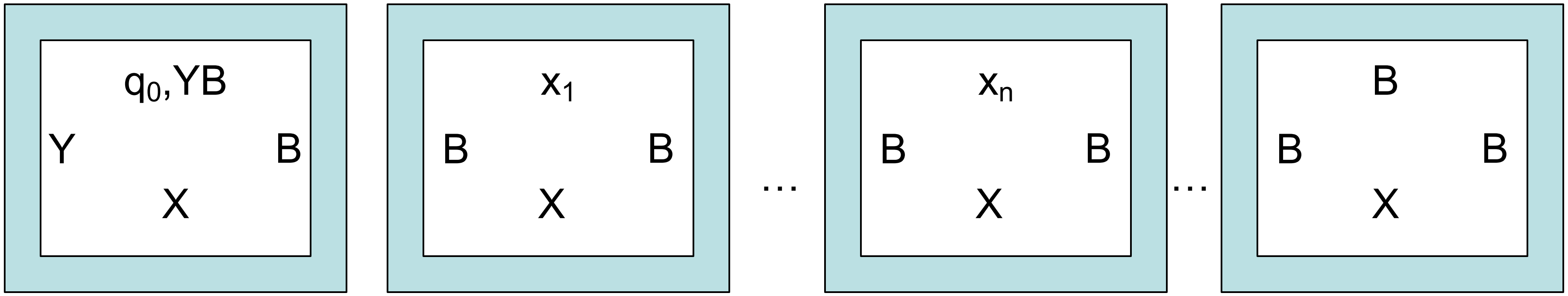
- The starting blank tape is not a real constraint as we can create M so its first actions are to write arguments on its tape.
- The semi unbounded tape is not new. If you look back at Standard Turing Computing (STC), we assumed there that we never moved left of the blank preceding our first argument.
- If you prefer to consider all computation based on the STC model then we add to M the simple prologue $(R1)^{x1}R(R1)^{x2}R\ldots(R1)^{xk}R$ so the actual computation starts with a vector of x1 … xk on the tape and with the scanned square to the blank to right of this vector. The rest of the tape is blank.
- Think about how, in the preceding pages, you could actually start the tiling in this configuration.

Bounded Tiling Problem #1

- Consider a slight change to our machine M. First, it is non-deterministic, so our transition function maps to sets.
- Second, we add two auxiliary states $\{q_a, q_r\}$, where q_a is our only accept state and q_r is our only reject state.
- We make it so the reject state has no successor states, but the accept state always transitions back to itself rewriting the scanned square unchanged.
- We also assume our machine accepts or rejects in at most n^k steps, where n is the length of its starting input which is written immediately to the right of the initial scanned square.

Bounded Tiling Problem #2

- We limit our rows and column to be of size n^k+1. We change our initial condition of the tape to start with the input to M. Thus, it looks like



- Note that there are n^k – n of these blank representations at the end. But we really only need the first.

Bounded Tiling Problem #3

- The finitely bounded Tiling Problem we just described mimics the operation of any given polynomially-bounded non-deterministic Turing machine.
- This machine can tile the finite plane of size $(n^k+1) * (n^k+1)$ just in case the initial string is accepted in n^k or fewer steps on some path.
- If the string is not accepted then we will hit a reject state on all paths and never complete tiling.
- This shows that the bounded tiling problem is NP-Hard
- Is it in NP? Yes. How? Well, we can be shown a tiling (posed solution takes space polynomial in n) and check it for completeness and consistency (this takes linear time in terms of proposed solution). Thus, we can verify the solution in time polynomial in n.

A Final Comment on Tiling

- If you look back at the unbounded version, you can see that we could have simulated a non-deterministic Turing machine there, but it would have had the problem that the plane would be tiled if any of the non-deterministic choices diverged and that is not what we desired.
- However, we need to use a non-deterministic machine for the finite case as we made this so it tiled iff some path led to acceptance. If all lead to rejection, we get stalled out on all paths as the reject state can go nowhere.

Comments on Variations

- One dimensional space (think about it)
- Infinite 3d space (really impossible in general)
  - This become a ∀∃ problem
  - In fact, one can mimic acceptance on all inputs here, meaning M is an algorithm iff we can tile the 3d space