A bird's eye view

Foundatioal tools

Incrementally adding features

Advanced topics

Date	Topic
Week 1	**How to design a programming language?**
06/24	Why study programming languages? + Python review
06/25	Syntax
06/26	Inference rules
Week 2	**What makes a programming language?**
07/01	Semantics
07/02	Names
07/03	Types
Week 3	**How to abstract *data*?**
07/08	Finite and recursive types
07/09	Pattern-matching
07/10	*Quiz 1 (tentative)*
Week 4	**How to abstract *computation*?**
07/15	Lambda calculus
07/16	Polymorphism, type inference
07/17	Defunctionalization, continuation-passing
Week 5	**How to change the world?**
07/22	Mutable states
07/23	Effect handlers
07/24	*Quiz 2 (tentative)*
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD
08/02	**(End of summer session A)**

Review - match each concept on the left with an item on the right

Relation

(x, y, z) is a member of relation R

Size of relation R

Arity of relation R

Judgment

Google sheets table

Number of columns

A statement about relation membership

Table R has a row (x, y, z)

Number of rows

A note about judgment notation

Example

(x, y) is a member of $\underbrace{\text{binary}}_{\text{arity} = 2}$ relation R

"$\sim$"

$R(x, y)$

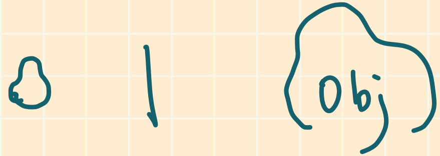
$\sim(x, y)$

$x \, R \, y$

$x \sim y$

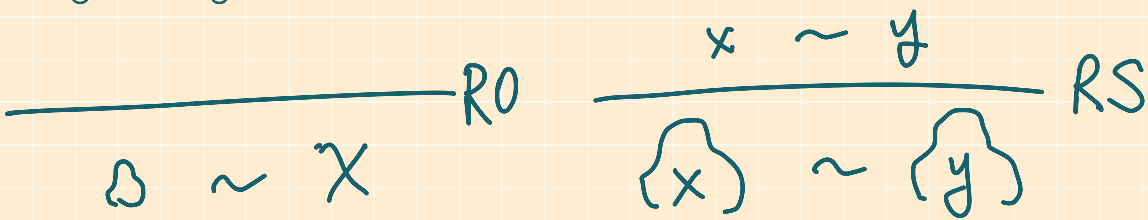
Each "column" in a relation contains some sort of objects
In CS 162, we'll describe objects using CFG.

Example:
"_ ~ _" is a binary relation with columns.
Objects in both columns are described by the CFG:

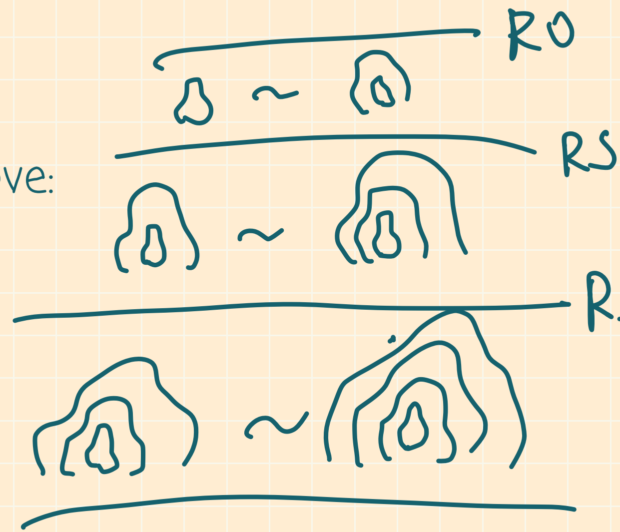
Obj ::= 

Judgment: Obj1 ~ Obj2

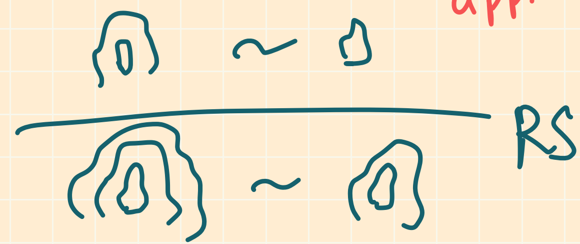
Rules:



Let's prove:

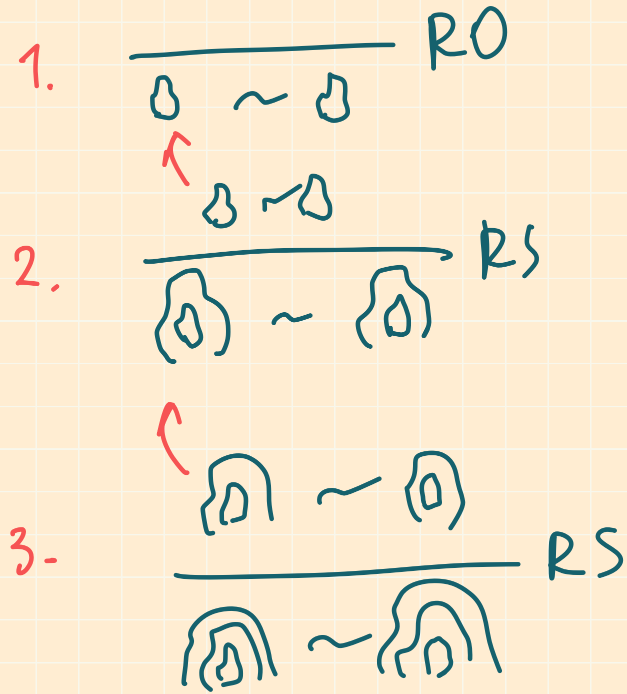


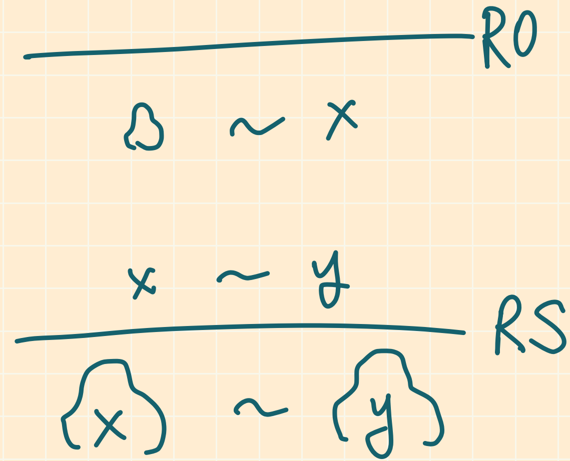
"stuck" if no rule applies



Claim: For any object x, we can show x~x.
In other words, "~" is a reflexive relation.

Proof:





Claim: For any object x, we can show x~x.
In other words, "~" is a reflexive relation.

Induction: a recursive algorithm to compute a proof of Claim(n) for any natural number n.

```
def induct(n):
  if n == 0:
    return base_case
  if n > 0:
    smaller_proof = induct(n-1)
    bigger_proof  = grow(smaller_proof, n)
    return bigger_proof
```

If you tell me how to implement `base_case` and `grow`, `induct` can automatically build a proof for any n.

```
def induct(n):
  if n == 0:
    return base_case
  if n > 0:
    smaller_proof = induct(n-1)
    bigger_proof  = grow(smaller_proof)
    return bigger_proof
```

Claim: For any object x, we can show x~x.

To use induct, we need a natural number metric to induct on.

aka "measure"

Recall objects are defined by the CFG:

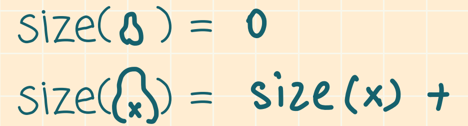
We can define a size metric/measure

size(Δ) = 0

size(x) = size(x) +1

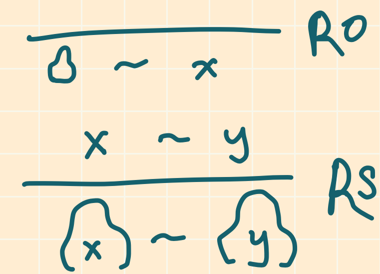
Claim(n): For any natural number n, if size(x) = n, we can show x~x.

```python
def induct(n):
  if n == 0:
    return base_case
  if n > 0:
    smaller_proof = induct(n-1)
    bigger_proof  = grow(smaller_proof)
    return bigger_proof
```

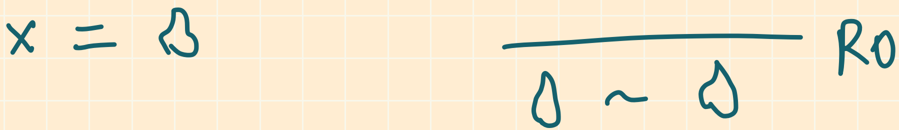


Claim(n): for any x, if size(x) = n, we can show x~x.

Rules:

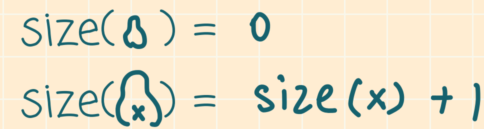


Base case: for all x, size(x) = n = 0, want to show x ~ x



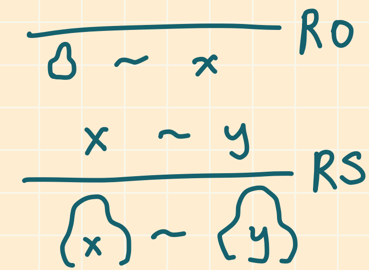
Grow(smaller_proof, n)

```python
def induct(n):
  if n == 0:
    return base_case
  if n > 0:
    smaller_proof = induct(n-1)
    bigger_proof  = grow(smaller_proof)
    return bigger_proof
```



Claim(n): for any x, if size(x) = n, we can show x~x.

Rules:

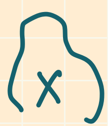


Grow(smaller_proof) where n > 0

smaller_proof says: for any x, if size(x) = n-1, then x~x.

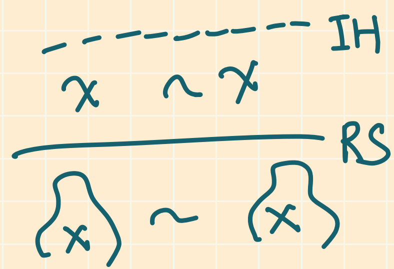
we want: for any y, if size(y) = n, then y~y

n>0

y =  size(x) = n-1

By IH, x ~ x.

y ~ y ?



means we're stacking another proof called IH (not a rule)

Exercise: show that for all x, if $x \sim$, then x must be .

$\text{size}() = 0$

$\text{size}() = \text{size}(x) + 1$

Step 1: choose a natural number metric n.

Rules:

$$\frac{}{\ \sim x}\ \text{LO} \qquad \frac{x \sim y}{\ \sim\ }\ \text{LS}$$

Step 2: rephrase the claim using n.

Step 3: base case: prove claim when n = 0.

Step 4: grow: assume we know claim holds for n-1, show we can grow the proof to n.

(Take-home) exercise: prove that ~ is transitive.

For all x y z, if x~y, y~z, then x~z.

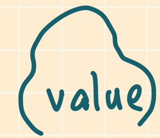
Hint: pick the right x, y, or z to induct on.

Even more Russian dolls

$\texttt{expr} ::= \texttt{value}$

$\quad\quad\;\; \mid\; \texttt{expr} \otimes \texttt{expr}$

$\texttt{value} ::=$ 

$\quad\quad\;\; \mid$ 

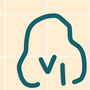
Judgment: $\texttt{expr} \Rightarrow \texttt{value}$

Rules:

$$\frac{}{v \Rightarrow v}\ \text{RV}$$

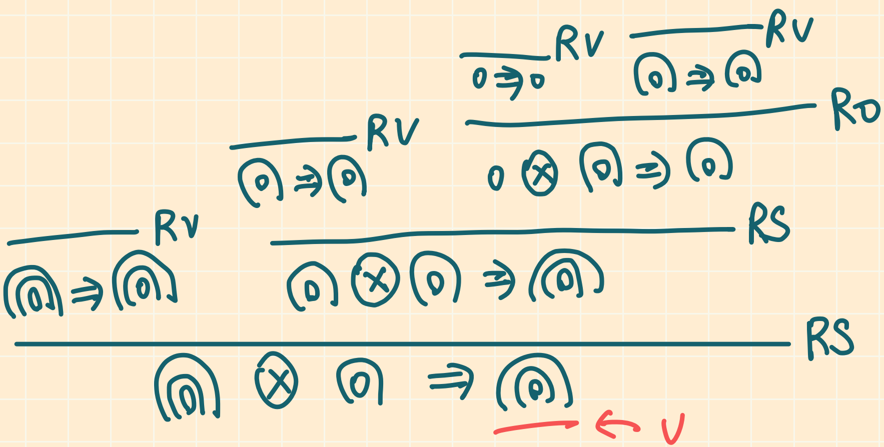
$e_1 \Rightarrow$  $\qquad e_2 \Rightarrow v_2$

$e_1 \otimes e_2 \Rightarrow v_2$ RO

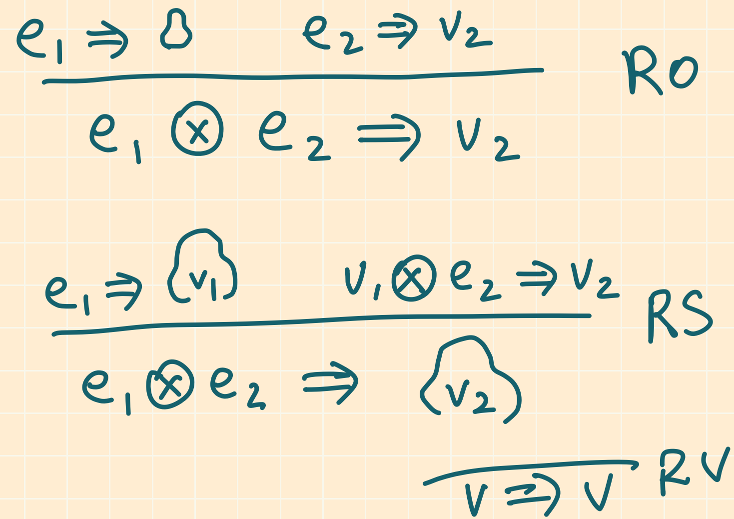
$e_1 \Rightarrow$  $\qquad v_1 \otimes e_2 \Rightarrow v_2$

$e_1 \otimes e_2 \Rightarrow$  RS

Claim: If I give you an expr e, you can naturally figure out a value v such that e $\Rightarrow$ v as you build the derivation tree.



Rules:



Let's find a value v such that 

Grand reveal

$$\text{value} ::= \quad \mid \ (\text{value}) \quad \longrightarrow \quad \text{Nat} ::= 0 \ \mid \ S(\text{Nat}) \quad \text{"successor"}$$

$$\text{expr} ::= \text{value} \ \mid \ \text{expr} \otimes \text{expr} \quad \longrightarrow \quad \text{expr} ::= \text{Nat} \ \mid \ \text{expr} + \text{expr}$$

$$\text{value} \sim \text{value} \quad \longrightarrow \quad \text{Nat} \leq \text{Nat}$$

Operational semantics: what a program returns eventually.

- `expr` = set of all programs
- `value` = set of all final results
- A judgment of the form `expr ⇒ value`
- Defined using inference rules

We just saw: the operational semantics of natural numbers with addition.

Augment the language with *.

Exercise: define the operational of *.

```
expr ::= value
       | expr + expr
       | expr * expr
```

```
value ::= 0
        | S(value)
```

Judgment: expr => value

Rules:

$$\frac{}{value \Rightarrow value}\ RV$$

$$\frac{e_1 \Rightarrow 0 \qquad e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow v_2}\ A0$$

$$\frac{e_1 \Rightarrow S(v_1) \qquad v_1 + e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow S(v_2)}\ AS$$

$$\frac{e_1 \Rightarrow 0 \qquad [e_2 \Rightarrow v_2]}{e_1 * e_2 \Rightarrow 0}\ M0$$

$v_1 * e_2 \Rightarrow v_2$

$$\frac{e_1 \Rightarrow S(v_1) \qquad [S(v_1) * e_2 \Rightarrow v_2]}{e_1 * e_2 \Rightarrow [v_2] \quad v_2 + e_2}\ MS$$

$$\frac{e_1 \Rightarrow 0 \quad [e_2 \Rightarrow v_2]}{e_1 * e_2 \Rightarrow 0}\ \text{M0}$$

Using your rules, prove S(0) * 0 => 0

$$\frac{e_1 \Rightarrow S(v_1) \quad S(v_1) * e_2 \Rightarrow v_2}{e_1 * e_2 \Rightarrow v_2}\ \text{MS v1}$$

doesn't terminate

$$\frac{e_1 \Rightarrow S(v_1) \quad v_1 * e_2 \Rightarrow v_2}{e_1 * e_2 \Rightarrow \underbrace{v_2 + e_2}}\ \text{MS v2}$$

"final value" still needs to be evaluated

$$\frac{e_1 \Rightarrow S(v_1) \quad v_1 * e_2 \Rightarrow v_2 \quad v_2 + e_2 \Rightarrow v_3}{e_1 * e_2 \Rightarrow v_3}\ \text{MS v3}$$

works

Have you used print for debugging?

CoinPython - a simplified Python language with coin flips (*)

```
msg  ::= “hi” | “sup”
prog ::= pass
       | raise
       | print(msg)
       | prog; prog
       | if (*) { prog } else { prog }
       | while (*) { prog }
```

CoinPython programs don't return a final value..

But we can still define an operational semantics for it

"The program P will terminate normally & print the string S"

"The program P will terminate normally & print the string S"

$$\boxed{P \Rightarrow S}$$

prog string

```
msg  ::= "hi" | "sup"
prog ::= pass
       | raise
       | print(msg)
       | prog; prog
       | if (*) { prog } else { prog }
       | while (*) { prog }
```

Rules:

$$\frac{}{\texttt{pass} \Rightarrow \text{" "}}\ \text{Pass}$$

(No rule for raise)

$$\frac{}{\texttt{print(msg)} \Rightarrow \texttt{msg}}\ \text{Print}$$

$$\frac{P_1 \Rightarrow S_1 \quad P_2 \Rightarrow S_2 \qquad (S_1 + S_2 = S_3)}{P_1 \,;\, P_2 \Rightarrow S_3}$$

← "side condition" to perform some uninteresting computation that we don't wanna model with rules.

(you could if you want)

$$\frac{P_1 \Rightarrow S_1}{\text{if } (*) \ \{ P_1 \} \ \text{else} \ \{ P_2 \}} \ \text{IfTrue}$$

$$\frac{P_2 \Rightarrow S_2}{\text{if } (*) \ \{ P_2 \} \ \text{else} \ \{ P_2 \}} \ \text{IfFalse}$$

$$\frac{}{\text{while } (*)\ \{\ p\ \} \Rightarrow \text{" "}}\ \text{WhileFalse}$$

$$\frac{?}{\text{while } (*)\ \{\ p\ \} \Rightarrow\ ?}\ \text{WhileTrue}$$