Functional programming

center of programming

no loop. no statement, just expression.

Arithmetic

$x + x * 3$

$= 2 + 6$

$= 8$

Let expression

let x = 2 x 3 in x + x

= let x = 6 in x + x

= 6 + 6 = 12

let x =
  let x = 2 x 3 in
    x + 1.
in x + 1

Pf: let x =
  let x = 6 in x + 1
in x + 1

= let x = 6 + 1 in x + 1.

= let x = 7 in x + 1

= x = 7 + 1 = x = 8

Variables

Let x = 2 * 3
  2 + x

Pf let x = 6 → substitute
  2 + x

= 2 + 6 = 8

let x =
  let y = 2 x 3 in y + 1
in x + y

Pf: let x = let y = 6 in y + 1
  in x + x

= let x = 6 + 1 in x + x

= let x = 7 in x + x

= 7 + 7 = 14

Lecture 2.

Arithmetic Expressions - syntax

expression: 1 x 2 + 3 x 3

1 is subexpression.
3 x 3 is subexpression.
+, x is operator.
3 x 3 is operand
3 is number literal

Semantic - how do you compute

Theorem 1 x 2 + 3 x 3 ≡ 19 ↓ value: expression that cannot be further reduced

proof
10 + 3 x 3 (arithmetic)
= 10 + 9 (arithmetic)
= 19 (arithmetic)

variable

bound expression of let expression.
sum expression.
let x = 5 x 2 in x + x ⇒ let expression.
↓ the bound variable
the body of the let expression.

pf: let x = 10 in x + x
≡ 10 + 10 x is bound to 10 (arithmetic)
(substitution of value of bound expression for use of bound variable in the body)
≡ 20 (arithmetic)

Key idea: variables are given meaning by substitution.

warning: "variables" in some language are not mathematical variable.

for example:
```
int x = 0;
for (x = 0; x < 10; x++)
```
substitute ⇒ for (x = 0) 0 < 10, 0++) always true

key idea: let expression are ordinary expressions so they can appear on body of another let expression

Scope
```
let x =
    let y = 1 x 2 in y + 1
in
    x + y    ?
```
⇒ not a valid value.

key idea: the scope of a variable bound by let is the body of that let

```
let x = 5 x 2 in
let y = x + 1 in
let x = 3 x 3 in x + y
```
shadowing

PF
```
let x = 10
let y = x + 1 in
let x = 3 x 3 in x + y
```
→ x is free and no more definition

≡
```
let y = 10 + 1 in
let x = 3 x 3 in x + y
```

≡
```
let x = 9 in x + 11
```

Other type of data

Primitive Type.

Int	-42, -3, 0, 3, 42, ...	+ - x / < >
Bool	true, false	&&, ||, if else

conditional expression.

if (x < 9) → guard. then 1 else (2) → branch.

Float	7.5, 0.3	+., x., -., /., <., >. (can't work with Int)

Expression have types. that predicts the type of expression's value

2+2 has type Int ⇒ 2+2 : Int ≡ 4 : Int (well typed)

(3 < 2) + 1 ⇒ has no type (ill-typed), ⇒ stuck

Type safety : well typed expressions don't go wrong

1. don't get stuck ←
2. don't change their type as evaluation.

Product / Tuples (compound values)

(1, 2) : Int × Int or Int, Int

(2+3, 3+3) ↑

(true, 1) : Bool × Int

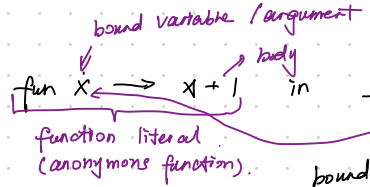
(true, 1, false, 3.5) : Bool × Int × Bool × Float

How to project out the elements of a tuple

let x=(a, b, c, d) - ?

type x = Bool × Int × Bool × Float

Functions



PF: let f = fun x → x+1 in f(5).
bound expression

★ functions are values ⇒ (cannot be separate)
value of f is the function.

≡ (fun x → x+1) (5) (substitution for x).
(substitute value of argument 5, for argument variable x, in body x+1)

≡ 5+1 (arithmetic).

≡ 6

let y : (Int, Int, Int) → Int =

fun mxb → let (m, x, b) = mxb in m*x + b

or syntax sugar.

fun (m, x, b) → m*x + b

Lab 1

- Values are expressions.
- Expressions are not necessarily values
  - evaluating to value doesn't mean it's a value

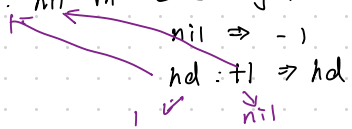
if true then a else false inconsistent branches (error) ⇒ have different type

Get value from tuple

```
let (a, b, _) = (2, true, 3.14) in
if b then a+1
else a
```

List

```
let my_list = nil in my_list   → empty list
1::[] = 1::nil = [1]
let my_list = 1::nil in case my_list of
                          nil ⇒ -1
                        | hd::tl ⇒ hd
```



Lecture 3.

```
let add5 : int -> int = fun n -> n+5 in
```

Design principle: orthogonality (naming is separate to function definition)

Recursion

```
let fac : int -> int =
    fun n ->
        if n <= 0 then 1 else n * fac(n-1)
in
fac(4)
```

= 24.

Key idea: Let is recursive if it has a function type annotation.

Scope of bound variable is both
- bound expression
- body

Exercise: `fib : int -> int`

List

Type: `[τ]`

Values: `nil` `Vhd::Vtl` (cons)

Operations:
```
case e,
| nil => e2
|Vhd:Vtl => e3
```

left: head right: tail

```
Pf  is_empty(mylist) ...        = false.
= case [1::2::5::nil]   "scrutinee"
    | nil => true
    | hd::tl => false
= false.
```

pf. length (1::2::5::nil) ≡ 3

≡ case (1::2::5::nil)
  | nil => 0
  | hd::tl => 1+ length(tl)

expression vs statement
vs value ?

≡ 1+ length (2::5::nil)

≡ 1+ case (1::2::nil)
  | nil => 0
  | hd::tl => 1+ length(tl))

≡ 1+ (1+ length(5::nil)) ... ≡ ... ≡ 1+(1+(1+ length(nil))
≡ 1+(1+(1+(1+ case nil ...)
= 1+(1+(1+0)

Dry principle: "Don't Repeat Yourself" for recursions ?

define a function => recursion scheme.

☆ right fold recursion: start from right, evaluate inside out.

```
let fold_right :  =
  fun (xs, b, f) →
    case xs
      | nil => b
      | hd::tl => f(hd, fold_right (tl, b, f))
```

```
let length = fun xs → fold_right (xs, 0, fun (hd, length_tl)
                                  → 1+ length_tl)
```

Tail recursion work remaining after all call is not tail recursion.

Compiler can implement more effectively.

Lecture 4. Equational and Inductive Reasoning.

fold_right (v1 :: v2 :: v3 :: nil, b, f)

= f (v1, fold_right (v2 :: v3 :: nil, b, f))

= ... f (v1, f (v2, f (v3, b))) ⇒ see v3 first.

If we choose f to be plus function

plus = fun (x, y) → x + y

so plus v1 v2 ≡ v1 + v2

then v1 + (v2 + (v3 + b))

Tail Recursion

When recursive call appears in return position. improve performance

↳ Can be optimized to have a constant sized stack.

```
let sum_plus : ([Int], Int) -> Int =
    fun (xs, acc) ->
        case xs
          |nil => acc
          |hd :: tl => sum_plus (tl, hd + acc)

let sum : [Int] -> Int =
    fun xs -> sum_plus (xs, 0) in . sum (1 :: 2 :: 3 :: nil)
```

PRY:

```
let fold_let =
    fun (xs, acc, f) ->
      case xs
        |nil => acc
        |hd :: tl => fold_left (tl, f (acc, hd), f).
      end
```

fold_left (v1 :: v2 :: v3 :: nil, b, f)

= fold_left (v2 :: v3 :: nil, f (b, v1), f)

= fold_left (v3 :: nil, f (f (b, v1), v2), f)

= fold_left (nil, f (f (f (b, v1), v2), v3), f)

= ↓ .

Proving Properties (Theorem) About Functional Programs

Tests : prove that a specific leads to specific output.

∀ : for all . prove Theorems involving for all

Theorem : ∀ lists, xs, length (xs) ≥ 0.

proof : induction.

List induction priciple : To prove a property (theorem) of the form

For all xs, P(xs)

It suffices to prove:

1. (Base case) P(nil).
2. (Inductive step) For all hd and tl, if we assume P(tl) , P(hd :: tl) should be shown

↑ induction hypothesis

∀ xs, length(xs) ≥ 0.

proof By list induction.

1. Base case. P(nil)

length(nil) ≥ 0

length(nil)

= case nil

| nil ⇒ 0

| hd :: tl ⇒ ...

= 0

0 ≥ 0 (by definition) .

2. Inductive step

Assume the induction hypothesis. P(tl), length(tl) ≥ 0

now we must show P(hd :: tl), length(hd :: tl) ≥ 0.

length(hd :: tl)

= case hd :: tl

| n :: l ⇒ 0

| hd :: tl ⇒ 1 + length(tl)

= 1 + length(tl)

It length(tl) ≥ 0

By the IH, there must be an integer n ≥ 0 such that length(tl) = n

1 + n ≥ 0 (by elementary reasoning)

```
let stutter : [int] -> [int] =
    fun (xs) ->
      case xs
      | nil => nil
      | hd :: tl => hd :: hd :: stutter(tl).
      end.
```

Theorem: $\forall xs,\ length(stutter(xs)) \equiv 2 * length(xs)$

Proof By List Induction

1. Base case $P(nil)$

$length(stutter(nil)) \equiv 2 * length(nil)$

$length(stutter(nil)) \equiv 2 * 0.$

$\equiv 0$

$length(nil) \equiv 0$

$0 \equiv 0$

2. Inductive step

Assume: $P(tl)$, $length(stutter(tl)) \equiv 2 * length(tl)$

Show: $P(hd :: tl)$ $length(stutter(hd :: tl)) \equiv 2 * length(hd :: tl)$

$length(hd :: hd :: stutter(tl)) \equiv$

$1 + length(hd :: stutter(tl)) \equiv$

$2 + length(stutter(tl)) \equiv 2 * (1 + length(tl))$

$2 + length(stutter(tl)) = 2 + 2\ length(tl)$

$2 + 2 * length(tl) \equiv$

Why prove properties about program? 为. 形式 prof 的好处

① practice mental reasoning.

② To actually prove properties about important program

Compcert : formally verified C compiler.
CakeML
(Yatin Manichor)
Project Everest - TLS

Automated Theorem Provers. (Proof Assistents)

↳ read proof as programs and check them for correctness.

↳ lean.

F*.

Coq

Lecture 5 Syntax

Now: learn how to define and implement (prototype) language.

Motivation:

- help become better programmer
- Programming language is anywhere
  - General Purpose language
  - Domain-Specific language.
    - → Graphic shades.
    - → Data analysis tools (MATLAB)
    - → pipelines.
    - → email filters
    - → Latex, MD
    - → Home automation.
    - → Cooking recipes
    - → sports playbooks

→ Math is a form of programming ⇒ proofs are programs.

Specification → constrains., what to do | e.g. tests. types. properties (theorems).

Implementations → how to do.

Example. Sorting

Specify. A function $s: [Int] \to [Int]$ is a sorting algorithm if.

1. $s(xs)$ is sorted ⇒ for every pair of adjacent elements, (a,b), we have $a \leq b$
2. every element in xs is also in $s(xs)$ with same number of times.
3. every element is $s(xs)$ is also in xs for same number of times.

Implementations:

mergesort, quicksort, insertion sort.

Programming language definitions are specifications, which can constrain many implementation

spec	impl
C	gcc. clang.
python	PyPy. Cpython
JS	V8. spidermonkey.

Common rhetoric:

"~~Python~~ is slow" ⇒ should be "CPython is slow"

Our first programming language: AL: arithmetic language.

AL definition.

1. syntax - what are programs structurely. (what they look like)
2. Semantics - what is the computational meaning of a program

Syntax table of AL.

Sort	metavariable.	::= (can be formal by)	Concrete Form	Structural / Abstract form.
Exp	e	::=	n (→ any number.)	NumLit [n] (∀ one num literal for every n.)
			I is math number	
			$\underline{I}$ is an AL number literal.	
			$e_1 + e_2$	Plus(e1, e2)
			$e_1 * e_2$	Time(e1, e2)
			-e	Neg(e)
			e !	Fac(e)
			(e)	e.

Identical. same structure. AL expression.

$\underline{2} * \underline{3}$ is Times(NumLit[2], NumLit[3]) ← is mathmetical form.

we assume an identification convection.

$\underline{2} * \underline{3} + \underline{1}$ → Plus (Times(NL[2], NL[3]), NL[1]).

or: Times (NumLit[2], Plus (NumLit[3], NumLit[1]))

Caution ⇒ Ambiguous syntax table ⇒ define operator precedence & associativity

Precedence and Associativity.

$(2 * 3) + 1$

Precedence and associativity essitially give rules for how to fully parentesice an expression in concrete form

Operator	Precedence	Associativity
binary operation: $e_1 + e_2$	2	or same, could be same if some assoc.
binary operation: $e_1 - e_2$	1	
binary operation: $e_1 * e_2$	3	
unary operation: $-e$	↗ higer precedence.	
unary operation: $e!$	bind more tightly.	

Algorithm.

① start with high precedure operators.

② for each occurance, parentesize against operands.

Lab 2.

Functions are values

Currying : let add = fun(x,y) -> x+y

or let add = fun x -> (fun y -> x+y)

High order functions: take function as argument, return a function, or both

functions which takes functions can be thought of as a kind of "functional design pattern"

Why use -

① reduce duplication.

② Abstract over base case

③ Complexity

④. Compose with other patterns leading to concise code.

While design patterns exist in any language., higher order function give us a powerful mechanism for abstraction

Recursion schemes.

A family of functional design patterns for iterating through.

Map

```
let zero_all : [Int] -> [Int] = fun xs ->
    case xs of
      | nil  => nil
      | hd::tl => 0 :: (zero_all (tl))
  in zero_all ( _____ )
```

```
let map : (Int -> Int) -> Int -> Int = fun f -> fun xs ->
=>    case xs of
        | nil  => nil
        | hd::tl -> f (hd) :: map (f) (tl)
  in let
```

```
let map : (Int -> Int) -> Int -> Int = fun f -> fun xs ->
  case xs of
  | nil => nil
  | hd::tl -> f(hd) :: map(f)(tl)
in let zero : (Int -> Int) = fun _ -> 0 in
   let zero_all : [Int] -> [Int] = map(zero)
in zero_all ( ——————>
```

Fold

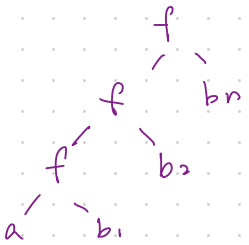
```
let foldr : ()->0->0->0 = fun f -> fun b -> fun xs ->
  case xs of
  | nil => b                    ? why not f(hd)::foldr(?)..
  | hd::tl => f(hd :: foldr(f)(b)(tl)).

let length : 0 -> 0 = foldr (fun(_,acc) -> 1+acc)(0) ; in ___
```

Fold left.

fold l f a (b1 :: ... bn :: nil).

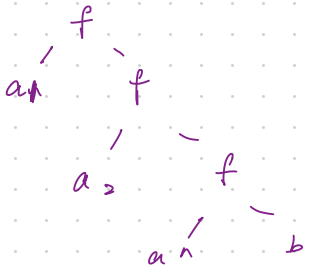
f(... (f(f a b1) b2) ...) bn.



Fold right.

foldr f b (a1 :: ... an :: nil)

f a1 (f a2 (... (f an b) ...))



Intuition behind Folds

Addition: fold_left / right do the same thing.

But exp is different, and subtraction is also different

Float in Hazel

Example

f (2nt, [2nt]) -> [2nt] .

```
let filter_even = fun xs ->
    fold_right ( fun (n, acc) ->
        if ==

    ) (n::l)
```

f : (2nt, [2nt]) -> [int]

```
f : fun x -> fun evens ->
        if x mod 2 == 0 then
            x :: even .

        else
            evens .
```

```
let foldr . = ( ) -> 0 -> 0 -> 0 = fun f -> fun b
-> fun xs -> case xs of
        | nil => b
        | hd :: tl => f(hd, foldr( f)( b) (tl) )
```

```
in
    let filter . 0 -> 0 -> 0 = fun pred -> foldr ( fun (x, acc)
-> if pred (x) then x :: acc else acc) (nil)
```

Compose - all. a function to compute list of function .

get input : [fun] -> [?]

? input of function ?

base case ?

```
let list_map = fun f -> fun xs -> case xs.
    | nil => nil
    | hd :: tl => f(hd) :: list_map(f, tl) end in
let fun_map = fun fs -> fun x in list_map (fun f -> f(x), fs)
```

Lab 2

Given
```
let concat (xs, ys) = case xs of
    | nil  => ys
    | hd::tl => hd :: concat (tl, ys)
let sum (xs) = case xs of
    | nil  => 0
    | hd::tl => hd + sum (tl)
```

Prove for all xs, concat (xs, nil) == xs

$\forall$ xs, concat (xs, nil) == xs.

Base case: (xs ≡ nil) By symmetry and transitivity of equivalence.

concat (xs, nil) concat (nil, nil) == nil

≡ concat (nil, nil)

≡ nil

Inductive step : Inductive hypothesis: concat (tl, nil) == tl

concat (xs, nil)

(xs ≡ hd::tl) ≡ concat (hd::tl, nil)

≡ case hd::tl of

| nil => nil

| hd::tl => hd :: concat (tl, nil).

≡ hd :: concat (tl, nil) ≡ hd :: tl.

Example 2. $\forall$ xs, $\forall$ ys, sum(concat (xs, ys)) == sum(xs) + sum(ys)

Base case (xs ≡ nil)

LHS: sum (concat (nil, ys)) RHS: sum (nil) + sum (ys)

≡ sum (ys). ≡ 0 + sum (ys)

≡ sum (ys)

Inductive: IH: sum(concat (tl, ys)) == sum(tl) + sum (ys)

sum (hd::tl) + sum (ys)

≡ hd + sum (tl) + sum (ys)

LHS: sum (concat (hd::tl, ys))

≡ sum (hd :: concat (tl, ys))

≡ hd + sum (concat (tl, ys))

≡ hd + sum (tl) + sum (ys) by IH

Associativity and Precedence.

Precedence

Determine the order you apply operations adjacent to each other

Matters when operators have different precedence

Associativity → how operators of same precedence are applied in an expression

((1+2)+3) + is left associative.

(1+(2+3)) + is right associative.

int → (float → bool) currying: we take single argument at a time

1::(2::(3::nil))

Fully parenthesize expressions

· put parenthesis around every top-level expression all the way down including child expressions and type annotations, but not the individuals leaves such as numbers

(1+(2*3)) ✓ 1+(2*3) ✗

exercise

	P	A
+	1	right
*	2	left
-	3	N/A

P ↑ 优先级↑ 含在一起

(3+(2+1))

((1+-2)*3)*4.

((-(-4))*(-2))

Lecture 6 Semantics

Syntax of AL ← form of language

Syntax Table

Expr e ::= (has following forms) ← metavariable

sort name

sort name: refer to it expressions

meta: another kind of shortand.

Concrete form		(Abstract) Structural form
	parsing → ← pretty printing	
n		NumLit [n]
$e_1 + e_2$		Plus(e_1, e_2)
(e)		e

AL is a white space insensitive language. Whitespace is secondary notion.

Ambiguous as specified ← motivation for precedence and associativity.

$\underline{2} * \underline{2} + \underline{5} * \underline{6}$

↳ $(((\underline{2} * \underline{2}) + \underline{5}) * \underline{6})$

Precedence and Associativity.

	Precedence	Associativity.
$e_1 + e_2$	2	left
$e_1 - e_2$	2	left
$e_1 * e_2$	3	left
$-e$	4	left
$e!$	5	left

(same precedence should have same associativity)

Example.

$(\underline{2} * \underline{2}) + (\underline{5} * (\underline{6}\ !))$ ⇒ Plus(Times(NLit[2], NLit[2]), Times(NLit[5], Fac[NLit[6]]))

Algorithm: (full parenthesization)

1. Go to the next highest precedence
2. Find the next operator, according to associativity, at that precedence in your input.
3. Parenthesize the adjacent operands, forming a single operand.
4. Repeat 2-3 until no operators at that precedence remain
5. Go to step one until no precedence remain

Another example

$(((2 + (3 * 4)) + (5 * 6)) - (-7))$

Example of right associative operators.

$((1::2)::3)::nil$ → left associativity

$1::(2::(3::nil))$ → right associativity

Implementing a syntax

parser : Text → Expr ← is there always an expr for every input text?

No

This is a partial function (undefined for some inputs) rather than a total function.

```
type Expr =
  | NumLit of (Int)
  | Plus (Expr, Expr)
  | Times (Expr, Expr)
  | Minus (Expr, Expr)
  | Neg (Expr)
  | Fac (Expr)
```

How to implement parsers?

→ Hand rolling (write it yourself, following simple algo)

→ Parser generators (synthesize parser from syntax spec)

→ Parser combinators.

Semantics ← computational meaning of syntactic structures.

A set of logical rules that give meaning to a syntax

We organize rules into judgements.

A judgement is a syntactic relation between one or more syntactic structures defined by a set of logical rules.

judgements

`e val` "e is a (an AL) value"

Rule (V-NumLit)

For all n, (we can derive) NumLit[n] val

Theorem: NumLit [17] val.

Proof By rule (V-NumLit), taking n to be 17

Theorem 17 val (recall identification convention: allow us to freely use concrete syntax and structural syntax implicitly, not using parse call)

Proof: same ↑

$e_1 \Downarrow e_2$ "e_1 evaluates to e_2" "e_1 has value e_2"

Rule: (E-NumLit)

for all n, $\underline{n} \Downarrow \underline{n}$

(E-Plus)

For all e_1, and e_2, and n_1, and n_2

if $e_1 \Downarrow \underline{n_1}$ and $e_2 \Downarrow \underline{n_2}$

and $n_1 + n_2 = n$. ← mathematical plus and n

$e_1 + e_2 \Downarrow \underline{n}$ from mathematical to AL plus

Theorem: $\underline{2} + \underline{2} \Downarrow 4$

Proof: By rule (E-plus), choosing $e_1 = \underline{2}$ $e_2 = \underline{2}$ $n_2 = 2$ $n_1 = 2$ $n = 4$.

To show 1) $\underline{2} \Downarrow \underline{2}$ (By rule (E-NumLit))

2) $\underline{2} \Downarrow \underline{2}$ (By rule (E-NumLit))

3) $2 + 2 = 4$ (By ambient math)

Inference rule notation (Natural Deduction notation) · syntax sugar

premise $\frac{J_1 \; J_2 \; \cdots J_n}{J}$ (name of rule) → If J_1 and J_2 and J_n then J.

conclusion judgement forms.

Implicitly: All of the meta variables that appear in $J_1 .. J_n$ and J are universally quantified at the top level of the rule.

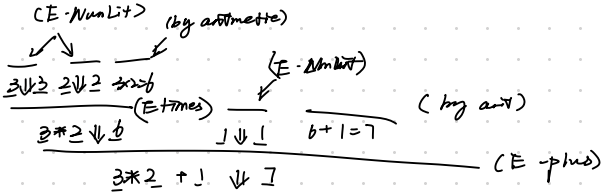
e val

$\frac{}{n \text{ val}}$ (V-NumLit).

Theorem $\underline{3} * \underline{2} + \underline{1} \Downarrow \underline{7}$.

Proof

upper level should all be axiom



Lecture 7 Variable and Function

- A1+A2 tonight due next Thursday.

Review

judgement form

AL Evaluation Semantics

e val "e is a value"

$$\frac{}{\underline{n}\ \text{val}}\ (\text{V-NumLit})$$

rule. the only val rule.

$\underline{n}$ is a val for all n

Theorem ⊢ val

Proof: By (V-NumLit), taking $n = 17$

use a rule to make judgement

$e_1 \Downarrow e_2$ or $e \Downarrow v$

$$\frac{}{\underline{n} \Downarrow \underline{n}}\ (\text{E-NumLit})$$

$\underline{n}$ evaluate to itself

judgement above the line need to justify below

and

$$\frac{e_1 \Downarrow \underline{n_1} \quad e_2 \Downarrow \underline{n_2} \quad n_1 + n_2 = n}{e_1 + e_2 \Downarrow \underline{n}}\ (\text{E-Plus})$$

$n_1 + n_2 = n$ ← mathmetic addition.

$$\frac{e_1 \Downarrow \underline{n_1} \quad e_2 \Downarrow \underline{n_2} \quad n_1 * n_2 = n}{e_1 * e_2 \Downarrow \underline{n}}\ (\text{E-Times})$$

Derivation

$$\frac{\dfrac{\dfrac{}{\underline{2} \Downarrow \underline{2}}\ (\text{E-NumLit}) \quad \dfrac{}{\underline{3} \Downarrow \underline{3}}\ (\text{E-NumLit}) \quad 2*3=6}{\underline{2}*\underline{3} \Downarrow \underline{6}}\ (\text{E-Times}) \quad \dfrac{}{\underline{1} \Downarrow \underline{1}}\ (\text{E-N}) \quad 6+1=7\ (\text{arithmetic})}{(\underline{2}*\underline{3})+\underline{1} \Downarrow \underline{7}}\ (\text{E-Plus})$$

$\underline{n}$ evaluate to itself

(E-N)

(arithmetic)

(by arithmetic)

(E-NumLit)

(E-Times)

(E-Plus)

不要写上所有的上划线.

一个 citation.

call stack?

Second Programming Language: ALL = AL + let Expression

we define syntax and semantics for a language.

Syntax:

	Concrete	Structual
Exp e, v ::=	... →	... → extending AL
	let x be e_1 in e_2	Let (x, e_1, e_2)
		or
		Let (e_1, x, e_2)
	x	Var[x]

bound binding

bound expression

body

x is in scope of e_2 / available in e_2

Evaluations Semantics.

$e\ \text{val}$ "e is an ALL value"

.... (expanding AL)

let expressions are never value ⇒ no rule for let, always some simplification to do.

variables are placeholds, not values, they have no meaning ⇒ no rule for variables.

↳ variable given meaning by substitutions.

Evaluation Judgement.

$e \Downarrow v$ "e evaluates to v"

... (expand all for AL)

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 = e_2' \quad e_2' \Downarrow v}{(\text{let } x \text{ be } e_1 \text{ in } e_2) \Downarrow v}$$

Substitute v_1 for free x in e_2

free in e_2 with isolation but bound by let

Example:

$$(\text{let } \underbrace{a}_{x} \text{ be } \underbrace{2*3+1}_{e_1} \text{ in } \underbrace{a*2}_{e_2}) \Downarrow 14$$

Proof: Let D be prev example.

$$\frac{\dfrac{(2*3)+1 \Downarrow 7 \quad [7/a](a*2) = 7*2 \quad 7*2 \Downarrow 14}{\text{let } a \text{ be } 2*3+1 \text{ in } a*2 \Downarrow 14}\ (\text{by } D) \qquad (\text{by definition of substitution})}{}\ (\text{E-let})$$

$$\frac{\dfrac{}{7 \Downarrow 7}\ (\text{E-N}) \quad \dfrac{}{2 \Downarrow 2}\ (\text{E-N}) \quad \dfrac{}{7+2=14}\ (\text{math})}{7*2 \Downarrow 14}\ (\text{E-times})$$

No value rule and evaluation rule for variable — given meaning by substitution

Not all ALL expressions have values.

let $x = 2+2$ in $x + y$ (x: bound variable; y: free variable (error))

An expression with no free variable is called a closed expression

Otherwise, open expression.

↳ Evaluation is only defined for closed expression.

Note: variables are given meaning → specify by substitution.

↳ Not necessarily implemented by substitution.

Third PL : ALL + functions (Lambda Calculus w/ Numbers) ALF

↑ Abstraction over value (Abstraction is hiding something and work on it)

Examples :

let plus1 = fun n → n+1 in plus1 (2+2) + plus1 (3*2+1)

Syntax

Exp e ::= ... (expanded) ...

fun x->e (argument binding, body) — Fun (x,e) or Fun (x.e) or λ x.e

e1 (e2) — Ap(e1, e2)

Lecture 8: Functions

Fuc cont.

ALF - ALL + Functions

Syntax

Exp e ::= ... ... (extended)

fun x → e ← body; argument binding

Fun (x, e)

or

λx.e

$e_1(e_2)$ or $e_1\ e_2$; e_1: function, e_2: argument

Ap (e_1, e_2)

space is binary operator with typically high precedence e.g (plus 1 2) * 3

and pick left-associativity ((f x) y) but not (f(x y))

↳ function without tuple: take one parameter at a time: currying.

Evaluation Semantics

$\boxed{e\ val}$ ... (extended from prev language)

$$\frac{}{fun\ x \to e\ \ val}\ (V\text{-}Fun)$$

axiom. all functions are values

Examples.

$$\frac{}{(fun\ x \to x+1)\ \ val}\ (V\text{-}Fun)$$

$$\frac{}{(fun\ x \to x+2*3)\ \ val}\ (V\text{-}Fun)$$

don't want to go in the body of function until an argument.

$\boxed{e \Downarrow v}$ ... (extended from prev language)

$$\frac{}{(fun\ x \to e) \Downarrow (fun\ x \to e)}\ (E\text{-}Fun)$$

also an axiom

or

$$\frac{}{e \Downarrow e}\ (E\text{-}Val)$$

$$\frac{e_1 \Downarrow (fun\ x \to e_{body})\quad e_2 \Downarrow v_2\quad ([v_2/x]\, e_{body}) \Downarrow v}{e_1(e_2) \Downarrow v}\ (E\text{-}Ap)$$

free variable.

Church Turing Theorem. ↓ these 2 can handle all

example.

$$\frac{\dfrac{}{(\text{fun } y \to y+1) \Downarrow (\text{fun } y \to y+1)}\,(\text{V-Fun}) \quad \dfrac{}{2 \Downarrow 2}\,(\text{V-Num}) \quad D}{(\text{fun } y \to y+1)(2) \Downarrow 3}$$

where $D =$

$$\frac{\dfrac{\dfrac{}{2 \text{ val}}\,(\text{V-N})}{2 \Downarrow 2}\,(\text{2-val}) \quad \dfrac{\dfrac{}{1 \text{ val}}\,(\text{V-N})}{1 \Downarrow 1}\,(\text{2-val}) \quad \dfrac{}{2+1=3}\,\text{(arith)}}{2+1 \Downarrow 3}$$

Equational Semantics

$e_1 \equiv e_2$ "e_1 is equivalent to e_2" → not necessarily a value

$$\frac{}{e \equiv e}\,(\text{Refl}) \quad \text{← axiom} \qquad \frac{e_2 \equiv e_1}{e_1 \equiv e_2}\,(\text{Sym.}) \qquad \frac{e_1 \equiv e_2 \quad e_2 \equiv e_3}{e_1 \equiv e_3}\,(\text{Transitivity})$$

Evaluation is not reflective. $2+2 \not\Downarrow 2+2$

Evaluation is not symmetric $2+2 \Downarrow 4$ but $4 \not\Downarrow 2+2$

Evaluation is transitive $2+2 \Downarrow 4 \quad 4 \Downarrow 4$

+ rule for ALF.

$$\frac{e_1 \equiv n_1 \quad e_2 \equiv n_2 \quad n_1 + n_2 = n}{e_1 + e_2 \equiv n}\,(\text{Eq-plus})$$

$$\frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{e_1 + e_2 \equiv e_1' + e_2'}\,(\text{Eq-plus-Cong})$$

apply equivalence anywhere you see for both sides

$$\frac{e = e'}{(\text{fun } x \to e) \equiv (\text{fun } x \to e')}\,(\text{Eq-Fun-Cong})$$

$$\frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{e_1(e_2) \equiv e_1'(e_2')}\,(\text{Eq-Ap-Cong})$$

$$\frac{}{(\text{fun } x \to e_{body})(e_2) \equiv [e_2/x]\, e_{body}}\,(\text{Eq-Ap}) \quad (\beta\text{-equivalence})$$

$$\frac{e \equiv [x/y]\, e'}{(\text{fun } x \to e) \equiv (\text{fun } y \to e')}\,(\alpha\text{-equivalence})$$

$$\frac{}{(\text{fun } x \to e) \equiv (\text{fun } y \to (\text{fun } x \to e)(y))}\,(\eta\text{-equivalence})$$

→ β-expansion

← η-reduction

$\text{fun } x \to f(x) \equiv f$.

Theorem let f = fun x → x + 1 in f(2) ≡ 3

Proof ≡ (fun x → x + 1)(2) ⟵ by β-equivalence.

≡ 2 + 1

≡ 3

Meta Theorem: If e ⇓ v then e ≡ v.

Theorem: ALF (λ calculus) can express any computable function (an expression as any programming language) ↔ Turing Completness // Universality.

Currying

let y = (fun m → (fun x → (fun b → m * x + b)))

in y 2

partial application

evaluate to a function ≡ fun x → fun b → 2 * x + b.

in ((y 2) 3) ≡ fun b → 2 * 3 + b

in (((y 2) 3) 1 ≡ 2 * 3 + 1 ≡ 7

↓ space is left associative

in y 2 3 1 y(2)(3)(1)

Partial Application without currying.

fun x → fun b → y(3, x, b) · more verbose and less flexiable with tuple

Lecture 9. Types

A type is a classifier of expressions that predicts the type of the expression's value (at compile time, statically.)

$\underline{2} : Num$ (has type) $\quad \underline{2} + \underline{2} : Num$

ALF_B - ALF + Booleans (have function and Boolean now)

Syntax	Concrete	Structural
Expr $e, v ::=$	. . .	. . . (extending ALF)
	true	True.
	false	False
	(if e_1 then e_2 else e_3) (guard)	If(e_1, e_2, e_3)

Evaluation Semantics

$$\frac{}{v \text{ val}} \quad \ldots$$

$$\frac{}{\text{true val}}\ (V\text{-true}) \qquad \frac{}{\text{false val}}\ (V\text{-false})$$

$$\frac{}{e \Downarrow v} \quad \ldots$$

(true and false evaluate via E-Val)

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v_2}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_2}\ (E\text{-If-True})$$

$$\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3}\ (E\text{-If-False})$$

$$\frac{}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow e_2}$$ (not a value yet) X since violates the Evaluation theorem

Theorem (Evaluation): If $e \Downarrow v$ then v val.

Type system

A type is a classifier of expressions that predicts the type of its value

Example: (let f = fun x → if x > 1 then 0 else 2 in f(3 + 2))

e : Num

so if $e \Downarrow v$ we know v: Num, i.e. the value will be a number

Static Semantics ← compile time a.k.a Type System

text —parse→ program structure —type checked→ well-typed structure —executing→ value.

compile time / static phase | run time / dynamic phase

Try #1:

$e : \tau$ "e has type τ"

Type J ::= Num | Bool | $\tau_{in} \to \tau_{out}$

Num | Bool | Arrow(τ_{in}, τ_{out}) ← syntax

$$\frac{}{\Gamma \vdash n : \text{Num}}\ (T\text{-NumLit}) \qquad \frac{}{\Gamma \vdash \text{true} : \text{Bool}}\ (T\text{-True}) \qquad \frac{}{\Gamma \vdash \text{false} : \text{Bool}}\ (T\text{-False})$$

$$\frac{\Gamma \vdash e_1 : \text{Num} \quad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash (e_1 + e_2) : \text{Num}}\ (T\text{-Plus})$$ (similar for multiplication, etc)

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}\ (T\text{-If})$$ ← e_2 and e_3 have same type

$$\frac{}{x \,.\, ?}\ (T\text{-Var})$$: need variables to have a type determined by their binding

Need a judgment that tracks variable and their types

$\Gamma \vdash e : \tau$ "e has type τ assuming Γ"

typing context (Gamma) | turnstile "entails"

typing contexts, Γ, are finite sets of assumptions.

$x_1 : \tau_1, \ldots, x_n : \tau_n$ $(n \geq 0)$ bind x to the value of e_1 for use of e_2

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (T\text{-Var}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}\ (T\text{-Let})$$ ← context extension

Example Derivation.

$$\frac{\dfrac{\dfrac{}{\cdot \vdash 2 : \text{Num}}\ (T\text{-NumLit}) \quad \dfrac{}{\cdot \vdash 2 : \text{Num}}\ (T\text{-NumLit})}{\cdot \vdash 2 + 2 : \text{Num}}\ (T\text{-Plus}) \quad \dfrac{\dfrac{y : \text{Num} \in y : \text{Num}}{y : \text{Num} \vdash y : \text{Num}}\ (T\text{-Var}) \quad \dfrac{}{y : \text{Num} \vdash 1 : \text{Num}}\ (T\text{-NumLit})}{y : \text{Num} \vdash y + 1 : \text{Num}}\ (T\text{-Plus})}{\cdot \vdash \text{let } y \text{ be } 2+2 \text{ in } y+1 : \text{Num}}\ (T\text{-Let})$$

(by definition) — why here.

y | e_1 | e_2 | τ_2

$$\frac{\Gamma, x : \tau_{in} \vdash e_{body} : \tau_{out}}{\Gamma \vdash (\text{fun } x \to e_{body}) : \tau_{in} \to \tau_{out}} \text{ (T-Fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_{in} \to \tau_{out} \qquad \Gamma \vdash e_2 : \tau_{in}}{\Gamma \vdash e_1(e_2) : \tau_{out}} \text{ (T-Ap)}$$

↙ type won't change after evaluation

Theorem (Preservation). if $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$.

Types of expression predicts types of values

Preservation is a key aspect of type safety.

Theorem (progress) if $\cdot \vdash e : \tau$ then e val or $e \Downarrow v$ (e for some v) ← recursion?

Implementing a type checker ☆

written in some compiler implementation language

(type analysis) $\underbrace{\Gamma}_{\text{input}} \vdash \underbrace{e}_{\text{input}} : \underbrace{\tau}_{\text{input}}$ type_ana : (Ctx, Expr, Typ) → Bool.

Theorem (Completeness)

if $\Gamma \vdash e : \tau$ then type_ana($\lfloor \Gamma \rfloor, \lfloor e \rfloor, \lfloor \tau \rfloor$) ≡ true. ($\lfloor \cdot \rfloor$: encoding of ...)

correct nets

$\lfloor \cdot \rfloor$ means "encoding of" or "implementation of" in implementation language

Theorem (Soundness)

if type_ana($\lfloor \Gamma \rfloor, \lfloor e \rfloor, \lfloor \tau \rfloor$) ≡ true then $\Gamma \vdash e : \tau$

$\underbrace{\Gamma}_{\text{input}} \vdash \underbrace{e}_{\text{input}} : \underbrace{\tau}_{\text{output}}$ (type synthesis)

Problem

- $\cdot \vdash \text{fun } x \to x : Num \to Num$
- $\cdot \vdash \text{fun } x \to x : Bool \to Bool$

⋮

Lecture 10. Products and Sums

Product types - classify values that group together multiple values

In Haskell: tuples

(3, true, 4) : (Int, Bool, Int)

3 tuple; product type

In OCaml: tuple: Bool * Bool * Bool

records: let c = { x : 3.5
y : 0.0
} : {
x : float
y : float.
}

c.x ⇒ 3.5

what is minimal ⇒

Type isomorphism.

$\tau_1 \cong \tau_2$ τ_1 is isomaphic to τ_2

if you can write two total functions:

into : $\tau_1 \to \tau_2$
out : $\tau_2 \to \tau_1$

where out (into (x)) ≡ x for all $x : \tau_1$
into (out (x)) ≡ x for all $x : \tau_2$

{x : float, y : float, z : float} ≅ float * float * float.

into : $\tau_1 \to \tau_2$ = fun(r) → (r.x, r.y, r.z)

out : $\tau_2 \to \tau_1$ = fun(x, y, z) → { x : x
y : y
z : z }

out (into({x : x, y : y, z : z})) ≡ {x : x, y : y, z : z}
into (out (x, y, z)) ≡ (x, y, z)

Claim: Any n-ary product (n ≥ 2) has an isomorphic product that requires only binary products

Ex. (int * float * (int * int * bool))
≅ (int * (float * ((int * int) * bool)) ↘ into.

So: we can restrict our interest formally to binary products.

ALF + Binary Products

Syntax

Expr $e ::=$ ... ... (extend ALF)

- introductory / value form: (e_1, e_2) — $Pair(e_1, e_2)$
- elimination forms:
  - $\text{let } (x,y) \text{ be } e_1 \text{ in } e_2$ — $Let(e_1, x.y.e_2)$ → destructuring
  - $e.0$ — $Prj\,0(e)$
  - $e.1$ — $Prj\,1(e)$ (projections)

Typ $\tau ::=$ ... ... (extend ALF)

$\tau_1 \times \tau_2$ — $Prod(\tau_1, \tau_2)$

Evaluation Semantics

$v \text{ val}$

$$\frac{v_1 \text{ val} \quad v_2 \text{ val}}{(v_1, v_2) \text{ val}} \text{ (V-Pair)}$$

no value rules for elimination rules

$e \Downarrow v$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \text{ (E-Pair)}$$

$$\frac{e \Downarrow (v_1, v_2)}{e.0 \Downarrow v_1} \text{ (E-Prj0)}$$

$$\frac{e \Downarrow (v_1, v_2)}{e.1 \Downarrow v_2} \text{ (E-Prj1)}$$

$$\frac{e_1 \Downarrow (v_1, v_2) \quad [v_1/x][v_2/y]e_2 \Downarrow v}{\text{let } (x,y) \text{ be } e_1 \text{ in } e_2 \Downarrow v} \text{ (E-letPair)}$$

$\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \text{ (T-Pair)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.0 : \tau_1} \text{ (T-prj1)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_2} \text{ (T-prjR)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x:\tau_1, y:\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{let } (x,y) \text{ be } e_1 \text{ in } e_2 : \tau} \text{ (T-LetPair)}$$

what about 1-tuples? no use of unary tuples.

$(\tau_1) \cong \tau$ → no need for 1-tuples.

what about nullary products? (0-tuples)

Yes, useful ⇒ next page

In Ocaml, unit type. () : unit.

Uses:

1. Deferring evaluation until / unless a condition is met.

```
let f = fun (flag : Bool, x : int) :
    if flag then g(x) else 0
in f(false, ackerman(100))
```

→ this is slow, waste of computational resource

→ calculate first.

Alternative:

```
let f = fun (flag : Bool, x : Unit → int) =
    ....
in f(false, fun () → ackerman(100))
```

→ delay until needed, since functions are values!

2. Define functions that only have side effects.

print : String → Unit

$$\frac{v \text{ val}}{() \text{ val}}\ (V\text{-Triv}) \qquad \frac{\Gamma \text{ e : t}}{\Gamma \vdash () : \text{Unit}}\ (CT\text{-Triv})$$

$$\underbrace{\text{Bool}}_{2} \times \underbrace{\text{Unit}}_{1} \cong \underbrace{\text{Bool}}_{2}$$

Sums are how you express choices / conditionals

```
type bool =
  | true
  | false
```

sum type; constructor

```
type planet =
  | Mercury
  | Venus
  ⋮
```

tick a → type parameter

```
type 'a option =
  | None
  | Some of 'a
```

```
let x : int option = None
let y : int option = Some(3)
```

Hazel:

```
case e
| p1 => e1
⋮
| pn => en
end.
```

Ocaml

```
match e with
| p1 => e1
⋮
| pn => en
```

```
let n : Bread : int =
  fun b ->
    case b
      | white => 0
      | wheat => 1
      | multi (n) => n
    end .
```

$bread' \cong \frac{bread}{1+1+\infty}$

```
type bread' = { is white : bool,
                is wheat : bool,
                is multi : bool,
                numgrain : int
              }
```

$2^3 \times \infty$

Exam: Open notes.

Lecture 11. Sum type

Sums are about expressing choices

```
type Bool =
  | True
  | False
```

```
type Option_ALFType
  | Some(ALFType).
  | none
```

if e1 then e2 else e3

```
case e
  | Some(ty) =>
  | none =>
```

type function.

```
type 'a option.=
  | Some 'a
  | none.
```

```
match e with
  | Some(x) ->
  | None ->
```

```
type ALFTyp
  | Num
  | Bool
  | Arrow (ALFTyp, ALFTyp)
  | Prod (ALFTyp, ALFTyp)
```

recursive!

What is the essential idea

Binary sums: every n-ary sum is isomorphic to some composition of binary sum.

How to simplify to binary sums?

1. It is always a binary choice.
2. The first choice is always called L, and the second is R
3. Every choices has to carry conditional data. ⇒ the unit to simplify.

Example:

```
type Bool =          type Bool' =
  | True      ≅        | L (Unit)
  | False              | R (Unit)
```

True → L(1)
False → R(1).
L(1) ← True.
R(1) ← False.

```
type Bread =
| White
| Wheat
| Mult (Num)
```

≃

```
type NonWhiteBread =
| L (Unit)
| R (Num)

type Bread' =
| L (unit)
| R (NonWhiteBread).
```

White ⟷ L(1)

Wheat ⟷ A(L(1))

Multi n ⟷ R(R(n)).

ALFA : ALF_p + Sum (binary) → choice ! (products + sums = "algebraic data types")

Syntax

Typ τ ::= ... ... extended

$\tau_L + \tau_R$ $Sum(\tau_L, \tau_R)$ ← choice between τ_L, τ_R

Examples.

Bread = Unit + (Unit + Num)
↑ L, ↑ R, R(L), R(R)

Bool = Unit + Unit = 1 + 1 (2)
↑ true, ↑ false

Option(τ) = Unit + τ = 1 + τ

Expr e ::= ...

Introductory form: L(e) | R(e)

... Inj L(e) | Inj R(e) ← Inject to choice

case e of
| L(x) ⇒ e_1
| R(y) ⇒ e_2

Case(e, x.e_1, y.e_2)

if e evaluate to L of v then x stands for v

→ elimination form

Evaluation Semantics

v val ... $\frac{v\ val}{L(v)\ val}$ (V-InjL)

$\frac{v\ val}{R(v)\ val}$ (V-InjR)

e ⇓ v ... $\frac{e \Downarrow v}{L(e) \Downarrow L(v)}$ (E-InjL)

$\frac{e \Downarrow v}{R(e) \Downarrow R(v)}$ (E-InjR)

Static Semantics (Type system)

$\frac{\Gamma \vdash e : \tau_L}{\Gamma \vdash L(e) : \tau_L + \tau_R}$ (T-InjL) cannot ensure ! type !

$\frac{\Gamma \vdash e : \tau_R}{\Gamma \vdash R(e) : \tau_L + \tau_R}$ (T-InjR)

Example: ⊢ L(2+2) : Num + Num
: Num + Bool
: Num + (Num → Num)

$$\frac{e \Downarrow L(v) \quad [y_x]e_1 \Downarrow v_1}{\text{case } e \text{ of } L(x) \to e_1 \text{ else } R(y) \to e_2 \Downarrow v_1} \text{ (E-Case-L)}$$

$$\frac{e \Downarrow R(v) \quad [y_y]e_2 \Downarrow v_2}{\text{case } e \text{ of } L(x) \to e_1 \text{ else } R(y) \to e_2 \Downarrow v_2} \text{ (E-Case-R)}$$

$$\frac{\Gamma \vdash e : \tau_L + \tau_R \quad \Gamma, x : \tau_L \vdash e_1 : \tau \quad \Gamma, y : \tau_R \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \text{ of } L(x) \to e_1 \text{ else } R(y) \to e_2 : \tau} \text{ (T-Case)}$$

(same!)

we don't know which branch will be take for runtime
⇒ only one rule, evaluate both at the same time

Example:

L R(L) R(R)
↓ ↓ ↓
Bread = 1 + (1 + Num)

let white : Bread be L(()) in

let wheat : Bread be R(L(())) in

let Multi : Num → Bread =

fun n → R(R(n)). in

let myFavBread = Multi (9) = R(R(9))

let num_grains : Bread → Num =

fun b →

case b of

L(x) → (unit) Unit ? But not useful. 0 (Num)

else R(y) → case y of

(1+Num) L(x) → 1 (unit)

else R(z) → z (num)

Lecture 12. Recursion

ALFA

```
let fac be
    fun n ->
        if n < 2 then 1 else n * fac (n - 1)
in
fac (4)
```

free variable: unbounded

scope for fac

OCaml

```
let rec fac =
    fun n ->
        if ...
in fac(4)
```

making it in scope for both

Attempt: add let rec to ALFA.

$$\frac{[?/x]e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v}{\text{let rec } x = e_1 \text{ in } e_2 \Downarrow}$$

- More general idea: self-referential expressions (fixpoints) → the essential idea of recursion

$\text{fix } x \to e$

x stands for this fixpoint itself in e

↳ let fac =
    fix fac → fun n → if n < 2 then 1 else n * fac (n - 1)
in fac (4)

fix point binding

let binding

the outer fac is not in scope inside ⇒ not shadowing, disjoint scope

syntax sugar: let rec x be e_1 in e_2 = let x be fix x → e_1 in e_2

How does $\text{fix } x \to e$ work

eval — fix x → e is never a value ⇒ different from functions.

...

unrolling the fixpoints

$e \Downarrow v$

$$\frac{[\text{fix } x \to e / x]e \Downarrow v}{\text{fix } x \to e \Downarrow v} \text{ (E-Fix)}$$

$e_1 \equiv e_2$

$$\frac{}{(\text{fix } x \to e) \equiv [\text{fix } x \to e / x]e} \text{ (Eq-Fix), axiom}$$

example. fix fac → fun n → if n < 2 then 1 else n * fac (n - 1)

e fix [...] body of fixpoint

≡ fun n → if n < 2 then 1 else n * (fix fac → fun n → if n < 2 then 1 else n * fac (n-1))(n-1)

now is a value since is a function!] V fix

apply to function

efix (4)

≡ vfix (4)

≡ if 4 < 2 then 1 else 4 * efix (4 - 1)

≡ if false then 4 * efix (4 - 1)

≡ 4 * efix (4 - 1)

≡ 4 * vfix (3)

≡ 4 * (if 3 < 2 then 1 else 3 * efix (3 - 1))

≡ 4 * 3 * efix (3 - 1) ≡ 4 * 3 * vfix (2)

≡ 4 * 3 * 2 * efix (2 - 1) ≡ 4 * 3 * 2 * vfix (1)

≡ 4 * 3 * 2 * (if 1 < 2 then 1 else 1 * efix (1 - 1))

≡ 4 * 3 * 2 * 1 ≡ 24

Mutual Recursion

let rec even = fun n → if n == 0 then true else odd (n - 1)

and odd = fun n → if n == 0 then false, else even (n-1)

AVFA + Fix define together with a pair !

let (even, odd) =

fix both →

(fun n → if n = ? 0 then true else both.1 (n-1)

, fun n → if n = ? 0 then false else both.0 (n-1)

)

in odd (5)

why you cannot destructure projection:

```
fix both ->
    let (even, odd) = both in
        (fun n ->    odd(n-2) ...
        ...
        )
```
efix

≡ `let (even, odd) = efix in (.., ..)`

self reference need to be inside.

`let syn : (Ctx, Expr) -> Option(Typ)` ← no known type.

`analysis : (Ctx, Expr, Typ) -> Bool` ← know the type (put type annotation)

(let: keyword in c++) (call mutual recursion)

ALFA without fixpoint is not universal (Turing Complete)

Ω combinator $(\lambda x.\ x\ x)(\lambda x.\ x\ x)$

≡ $(\lambda x.\ x\ x)(\lambda x.\ x\ x)$ evaluate to itself ⇒ forever loop, never terminate, never get value

Cannot give it a type

$(\lambda x : \sqcup \to \sqcup .\ x\ x)$

$\sqcup \to \sqcup$

$\sqcup \to \sqcup \to \cdots$

Define type for fixpoint

$\Gamma \vdash e : \tau$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix } x \to e : \tau}\ (\Gamma\text{-Fix})$$

STLC (simply-typed lambda calculus) + fixpoints ⇒ proven to be Turing complete

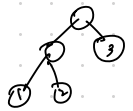
Although ALFA without fixpoint is not universal, it is normalizing.

every program terminates with / without value.

universality & normalization ⇒ can only have one fundamental trouble left.

```
type NumList =                          ← recursive type.
  | Nil    []  = Nil
  | Cons (Num, NumList)   hd::tl   = Cons(hd, tl)
```

```
type BTree =
  | Leaf (Num)
  | Node (BTree, BTree)
```



Node(Node(Leaf(1), Leaf(2)), Leaf(3)).

Lab 5

- bidirectional typing
- 2 modes { Synthesis, Analysis } recursively call each other

Synthesis "give me an expression and I'll tell you the (unique) type

- used when you don't have an expected type
- e.g. at top level, or when defining an unannotated let

The type synthesis judgement, $\Gamma \vdash e \Rightarrow \tau$, will be implemented by the syn function and can read "e synthesizes type τ assuming Γ"

Top level: $3+2 \Rightarrow$ Int

Unannotated Let: let $x = 2+3$ in x.

$2+3 \Rightarrow$ Int. $x \Rightarrow$ Int

Analysis "given expression and a type. Evaluate if the expression can be used where you need an expression of that type

The type analysis judgement, $\Gamma \vdash e \Leftarrow \mathcal{J}$, will be implemented by the ana function and can be pronounced "e analyzes against $\mathcal{J}$ assuming Γ"

- Annotated Let: let x : Int $= 2+3$

input output

$\Gamma \vdash e \Rightarrow \tau$ "e synthesizes type τ assuming Γ"

$\Gamma \vdash e \Leftarrow \tau$ "e analyzes (checks against) against τ assuming Γ"

syn > ana
since need less info and provide more

Subsumption (more powerful)

$$\frac{\Gamma \vdash e \Rightarrow \mathcal{J}}{\Gamma \vdash e \Leftarrow \mathcal{J}} \text{ (A-subsumption)}$$

e.g.

$$\frac{\dfrac{}{\Gamma \vdash \underline{n} \Rightarrow \text{Num}}\text{ (S-NumLiteral)}}{\Gamma \vdash \underline{n} \Leftarrow \text{Num}}\text{ (A-subsumption)}$$

Lecture 13

Recursion with level of type

```
type NumList =
  | Nil
  | Cons (Num, NumList)

type BTree =
  | Leaf (Num)
  | Node (BTree, BTree)
```

```
type ALFExpr =
  | NumLit (Num)
  | Plus (ALFExpr, ALFExpr)
  | ....
```

Abstract syntax tree

Add some fixpoint to type level for recursion.

```
type NumList =
  rec NumList is
    Nil
    + Cons (Num, NumList)
```

recursive sum

self-ref bound by "rec"

```
type NumList =
  (rec NumList is
    (1 (unit type)
    + (Num x NumList)))
```

```
type BTree =
  rec BTree is
    Num
    + BTree x BTree
```

convert to structural system

encode ALFA Typ as an ALFA type

Recursive types.

unroll to make it work

rec a is τ ≠ τ e.g.

roll: base for recursive.

roll (L(())) : NumList

L(()) ✗ : NumList

because NumList = rec NumList is 1 + Num × NumList.

has unrolling [NumList/NumList] (1 + Num + NumList)

= 1 + Num × (1 + Num × NumList)

rec NumList is

recursion is inside, not outside. sum is outside!

ALFA + Recursive Types.

Syntax

Typ τ ::= ... | rec t is τ (body) ... | t

self reference

(external ALFA)

Rec (t. τ)

TVar [t]

Expr $e ::= \ldots$ $\ldots$ (extends AUFA)

roll(e) Roll(e) ← intro form for rec

elimination form → unroll(e) Unroll(e)

Static Semantics (Type system).

$\boxed{\Gamma \vdash e : \tau}$

$\ldots$

$$\frac{\Gamma \vdash e : [\text{rec } t \text{ is } \tau_t]\,\tau}{\Gamma \vdash \text{roll}(e) : \text{rec } t \text{ is } \tau} \text{(T-Roll)}$$

$$\frac{\Gamma \vdash e : \text{rec } t \text{ is } \tau}{\Gamma \vdash \text{unroll}(e) : [\text{rec } t \text{ is } \tau / t]\,\tau} \text{(T-Unroll)}$$

Example:

$$\frac{\dfrac{\dfrac{\dfrac{}{\vdash () : 1}\text{(T-Triv)}}{\vdash L(()) : 1 + \text{Num} \times \text{NumList}}\text{(T-Inj L)}}{\vdash \text{roll}(L(())) : \underbrace{\text{rec NumList is } 1 + \text{Num} \times \text{NumList}}_{\text{NumList}}}\text{(T-Roll)}$$

(e: $\text{roll}(L(()))$; t: NumList; τ: $1 + \text{Num} \times \text{NumList}$)

let Nil : NumList = roll(L(())) in

let Cons : (Num, NumList) → NumList =

fun hdtl ⟶

let (hd, tl) be hdtl in

roll(R(hd, tl)) in.

Cons(3, Cons(4, Nil))

≡ roll(R(3, roll(R(4, roll(L(())))))) ← unit; the value

$$\frac{\dfrac{\dfrac{\dfrac{}{\vdash 4 : \text{Num}}\text{(T-Num)} \quad \dfrac{}{\vdash \text{roll}(L(())) : \text{NumList}}\text{(proved above)}}{\vdash (4, \text{roll}(L(()))) : \text{Num} \times \text{NumList}}\text{(T-Pair)}}{\vdash R(4, \text{roll}(L(()))) : 1 + \text{Num} \times \text{NumList}}\text{(T-Inj R)}}{\vdash \text{roll}(R(4, \text{roll}(L(())))) : \text{NumList}}\text{(T-Roll)}$$

rec Numl ...

let length : NumList → Num be

fix length → fun xs → case (unroll(e)) of

(either left injection → L(x) or right)

1 + Num × NumList

L(x) → 0

R(y) → 1 + length(y, 1)

NumList in Num × NumList, right projection

now good!

in length

e's type is recursive type

unroll(e) has a type of $1 + \text{Num} \times \text{NumList}$

One more complication:

$\text{rec } t \text{ is } 1 + \text{Num} \times t$ but syntaxly they are different for now

$\equiv \quad \text{rec } u \text{ is } 1 + \text{Num} \times u$

we need a rule ignore name!

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau_3 \qquad \tau_2 \equiv \tau_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

$\tau_1 \equiv \tau_2$

$$\frac{}{\tau \equiv \tau} \text{(Refl.)} \qquad \frac{\tau_2 \equiv \tau_1}{\tau_1 \equiv \tau_2} \text{(Sym)}$$

$$\frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3} \text{(transitivity)}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \text{(Cong - Arrow)}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'} \text{(Cong - Prod)}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 + \tau_2 \equiv \tau_1' + \tau_2'} \text{(Cong - Sum)}$$

$$\frac{\tau \equiv \tau'}{\text{rec } t \text{ is } \tau \equiv \text{rec } t \text{ is } \tau'} \text{(Cong - Rec)}$$

$$\frac{\tau \equiv [t/t']\tau'}{\text{rec } t \text{ is } \tau \equiv \text{rec } t' \text{ is } \tau} (\alpha - \text{Eq})$$

Method of de Bruijn Indices

Defining an entire α-equivalence class of types as an single structure:

$\text{rec } t \text{ is } 1 + \text{Num} \times t.$

$\text{rec } t \text{ is } 1 + \text{Num} \times u$

$\hookrightarrow \text{rec } \cdot \text{ is } 1 + \text{Num} \times \cdot$