Discussion 6

October 4

Logistics

- A4 due today at 8PM! Late deadline is Monday at 8PM
- A5 will be released later today
- Fill out midterm course evaluations!

Preview for today

- ALFA
  - Sum types
- Recursive ALFA
  - Fixpoints
  - Recursive types
- De Bruijn indices

Sum Types

Sums

- Corresponds to disjoint unions in discrete math

- In Hazel: `let val: (Type1 + Type2) = …`

- Allows you to pick an option between many types;
  think of it like OR for types, in contrast to AND for products

Sums: Examples

- Enums?

  (only if you squint; no type safety, no ability to combine types)

- Algebraic data types

  (combines products and sums)

OCaml

```
type number =
  | Int of int
  | Float of float

let x : number = Int 10 in
let y : number = Float 1.1 in
(x, y)
```

C

```
typedef enum {
  ARM_EXCEPTION_RESET = 0,
  ARM_EXCEPTION_UNDEF = 1,
  ARM_EXCEPTION_SWI = 2,
  ARM_EXCEPTION_PREF_ABORT = 3,
  ARM_EXCEPTION_DATA_ABORT = 4,
  ARM_EXCEPTION_RESERVED = 5,
  ARM_EXCEPTION_IRQ = 6,
  ARM_EXCEPTION_FIQ = 7,
  MAX_EXCEPTIONS = 8,
  ARM_EXCEPTION_MAKE_ENUM_32_BIT = 0xffffffff
} Arm_symbolic_exception_name;
```

typescript

```
type Alien = {
  type: 'unknown';
}

type Animal = {
  species: string;
  type: 'animal';
}

type Human = Omit<Animal, 'type'> & {
  name: string;
  type: 'human';
}

type User = Animal | Human;
```

hazel

```
type Exp =
  + Var(String)
  + Lam(String, Exp)
  + Ap(Exp, Exp) in
```

Examples of Sum Types (OCaml)

- Option

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

- Result

```
type ('t, 'e) result = Ok of 't | Error of 'e
```

- Lists

```
type 'a list = [] | :: of 'a * 'a list
```

Anonymous Binary Sums (ALFA)

- In general sum types can have any number of components
  `Type1 + Type2  + Type3 …`

- But for ease of implementation, we focus here on binary sums
  `Type1 + Type2`

- And to avoid having to implement a system to name our different cases (constructors), we will use anonymous binary sums which always use the constructors **L** & **R**

Binary Sums: Introduction & Elimination

- Introduction form: Left & Right Injections

```
L(e₁): Type1 + Type2                              (e₁: Type1)
R(e₂): Type1 + Type2                            (e₂: Type2)
```

- Elimination form: Case expressions

```
case e of                          (e: Type1 + Type2)
L(x) → … else                      (x: Type1)
R(y) → …                           (y: Type1)
```

Are anonymous binary sums enough?

- What if we want to pick from more than two kinds of thing?

Are anonymous binary sums enough?

- What if we want to pick from more than two kinds of thing?
- Remember how we build triples from pairs?
  We can do the same for sums.
- N-ary sums can be built up from binary sums, and named constructors can be defined by composing the anonymous constructors L and R

Building complex sums from simple sums

- Suppose we want to represent breakfast options. The choices are **Cereal**, **Oatmeal**, **Pancakes**, or **Omelet**. For the **Omelet**, we can choose how many eggs; for the **Pancakes**, whether we want berries

- How do we represent this using a sum type?
  (Let's start with an n-ary sum; then we'll convert to binary)

- We have four kinds of things to choose from,
  so let's use a 4-way sum

Building complex sums from simple sums

- `Breakfast ≡ ?? + ?? + ?? + ??`

Building complex sums from simple sums

- `Breakfast ≡ ?? + ?? + ?? + ??`
- Our first 2 components (cereal, oatmeal) only have one option each

  Which types are are appropriate?

Building complex sums from simple sums

- `Breakfast ≡ Unit + Unit + ?? + ??`

- Our third component can represent the Omelet option, where you can specify the number of eggs

  What type is appropriate?

Building complex sums from simple sums

- `Breakfast ≡ Unit + Unit + Num + ??`

- Our final component can represent the Pancakes option, which can be with or without berries

  What type is appropriate?

Building complex sums from simple sums

- `Breakfast ≡ Unit + Unit + Num + Bool`

- How do we make this a binary sum?

Building complex sums from simple sums

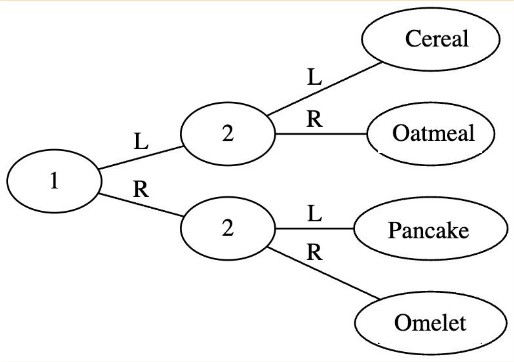
- `Breakfast ≡ Unit + Unit + Num + Bool`
- How do we make this a binary sum?
- Proposals:
  1. `Breakfast ≡ (Unit + Unit) + (Num + Bool)`
  2. `Breakfast ≡ ((Unit + Unit) + Num) + Bool`
  3. `Breakfast ≡ Unit + (Unit + (Num + Bool))`
- Doesn’t really matter. We’ll pick (1).

Constructors `(Unit + Unit) + (Num + Bool)`

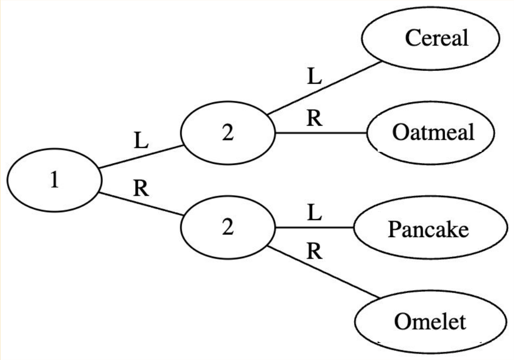
```
let cereal: Breakfast = ?? in
let oatmeal: Breakfast = ?? in
let pancake: Bool → Breakfast = ?? in
let omelet: Num → Breakfast = ?? in
```

Constructors `(Unit + Unit) + (Bool + Num)`

```
let cereal: Breakfast = ?? in
let oatmeal: Breakfast = ?? in
let pancake: Bool → Breakfast = ?? in
let omelet: Num → Breakfast = ?? in
```

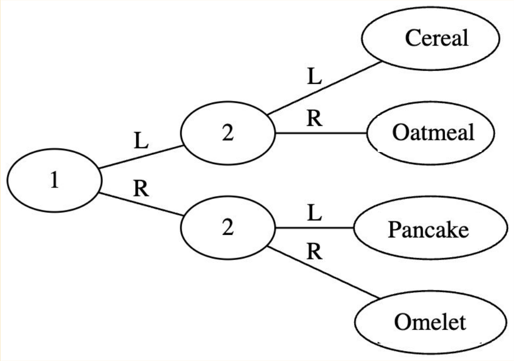


```
let cereal: Breakfast = L(L()) in
let oatmeal: Breakfast = ?? in
let pancake: Bool → Breakfast = ?? in
let omelet: Num → Breakfast = ?? in
```



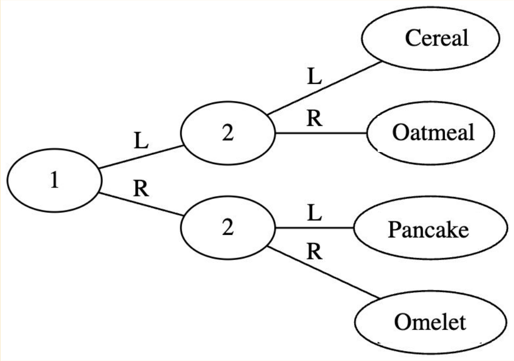
Constructors `(Unit + Unit) + (Num + Bool)`

```
let cereal: Breakfast = L(L()) in
let oatmeal: Breakfast = L(R()) in
let pancake: Bool -> Breakfast = ?? in
let omelet: Num -> Breakfast = ?? in
```



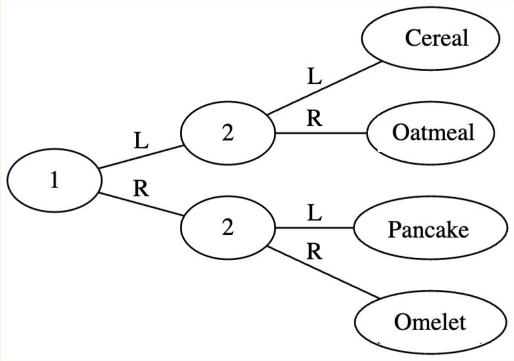
Constructors `(Unit + Unit) + (Num + Bool)`

```
let cereal: Breakfast = L(L()) in
let oatmeal: Breakfast = L(R()) in
let pancake: Bool → Breakfast = fun b → R(L(b)) in
let omelet: Num → Breakfast = ?? in
```



Constructors `(Unit + Unit) + (Num + Bool)`

```
let cereal: Breakfast = L(L()) in
let oatmeal: Breakfast = L(R()) in
let pancake: Bool → Breakfast = fun b → R(L(b)) in
let omelet: Num → Breakfast = fun n → R(R(n)) in
```



Using Proposal 1 `(Unit + Unit) + (Num + Bool)`

- Now let's write a function for the # eggs needed for a breakfast

```
let eggs_needed =
  ??
```

Using Proposal 1 `(Unit + Unit) + (Num + Bool)`

- Now let's write a function for the # eggs needed for a breakfast

```
let eggs_needed =
  fun b : (Unit + Unit) + (Num + Bool) →
    ??
```

Using Proposal 1 `(Unit + Unit) + (Num + Bool)`

- Now let's write a function for the # eggs needed for a breakfast

```
let eggs_needed =
  fun b : (Unit + Unit) + (Num + Bool) →
    case b of
      L(x) → ⁇
      else R(y) → ⁇
```

Using Proposal 1 `(Unit + Unit) + (Num + Bool)`

- Now let’s write a function for the # eggs needed for a breakfast

```
let eggs_needed =
  fun b : (Unit + Unit) + (Num + Bool) →
    case b of
      L(x) → 0
      else R(y) → ⁇
```

Using Proposal 1 `(Unit + Unit) + (Num + Bool)`

- Now let's write a function for the # eggs needed for a breakfast

```
let eggs_needed =
  fun b : (Unit + Unit) + (Num + Bool) →
    case b of
      L(x) → 0
      else R(y) →
        case b of
          L(x) → ??
          else R(y) → ??
```

Using Proposal 1 `(Unit + Unit) + (Num + Bool)`

- Now let's write a function for the # eggs needed for a breakfast

```
let eggs_needed =
  fun b : (Unit + Unit) + (Num + Bool) →
    case b of
      L(x) → 0
      else R(y) →
        case b of
          L(x) → x
          else R(y) → 0
```

Fixpoints

(non-termination for fun and profit)

Some motivation

Recall that we want to define recursive computations

```
let odd be
    fix odd is
        (fun (x : Num) ->
        if x =? 0 then False
        else if x =? 1 then True
        else odd(x - 2))
in ...
```

We can even get mutual recursion (fix on a pair)!

Static semantics of fix

- When the self-reference $x : \tau$ is incorporated into the typing context, the fixpoint body e has the type τ of the self-reference

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

Dynamic semantics of fix

- Replace all instances of the variable x with the fixpoint itself in the body e
  - In many cases, the body will be a function
- When evaluation reaches a recursive call, we'll unroll again to prepare for the next recursive call

$$\frac{[\mathrm{fix}(x:\tau) \to e/x]e \Downarrow v}{\mathrm{fix}(x:\tau) \to e \Downarrow v}$$

Exercise: prove odd(3) ≡ True

```
odd(3)
≡ (fix odd is fun (x : Num) ->                                      (E-Fix)
    if x =? 0 then False
    else if x =? 1 then True
    else odd(x - 2))(3)

≡ (fun (x : Num) ->                                                 (E-Ap)
    if x =? 0 then False
    else if x =? 1 then True
    else (fix odd is fun (x : Num) ->
              if x =? 0 then False
              else if x =? 1 then True
              else odd(x - 2))(x - 2)))(3)
```

Exercise: prove odd(3) ≡ True

```
≡ (fun (x : Num) ->                                  (E-Ap)
    if x =? 0 then False
    else if x =? 1 then True
    else (fix odd is fun (x : Num) ->
        if x =? 0 then False
        else if x =? 1 then True
        else odd(x - 2))(x - 2)))(3)

≡ (fix odd is fun (x : Num) ->                       (E-Fix)
        if x =? 0 then False
        else if x =? 1 then True
        else odd(x - 2))(1)
```

Exercise: prove odd(3) ≡ True

```
≡ (fix odd is fun (x : Num) ->                              (E-Fix)
        if x =? 0 then False
        else if x =? 1 then True
        else odd(x - 2))(1)

≡ (fun (x : Num) ->                                         (E-Ap)
    if x =? 0 then False
    else if x =? 1 then True
    else (fix odd is fun (x : Num) ->
              if x =? 0 then False
              else if x =? 1 then True
              else odd(x - 2))(x - 2)))(1)
```

Exercise: prove odd(3) ≡ True

```
≡ (fun (x : Num) ->                                        (E-Ap)
    if x =? 0 then False
    else if x =? 1 then True
    else (fix odd is fun (x : Num) ->
        if x =? 0 then False
        else if x =? 1 then True
        else odd(x - 2))(x - 2)))(1)

≡ True
```

Exercise: prove odd($\underline{3}$) $\equiv$ True

That was rather long and tedious—time to introduce some shorthands!

```
e_fun = fun (x : Num) ->
    if x =? 0 then False
    else if x =? 1 then True
    else e_odd(x - 2)

e_odd = fix odd is e_fun
```

Notice that e_{fun} and e_{odd} are *mutually recursive*!

Exercise: prove odd($\underline{3}$) ≡ True (shortened, full)

```
e_odd(3)                                   (E-Fix)
≡ e_fun(3)                                 (E-Ap)
≡ if 3 =? 0 then False                     (E-If-F)
  else if 3 =? 1 then True
  else e_odd(3 - 2)
≡ if 3 =? 1 then True                      (E-If-F)
  else e_odd(3 - 2)
≡ e_odd(3 - 2)                             (E-Minus)
≡ e_odd(1)                                 (E-Fix)
≡ e_fun(1)                                 (E-Ap)
≡ if 1 =? 0 then False                     (E-If-F)
  else if 1 =? 1 then True
  else e_odd(1 - 2)
≡ if 1 =? 1 then True                      (E-If-T)
  else e_odd(1 - 2)
≡ True
```

Exercise: prove odd($\underline{3}$) ≡ True (shortened)

```
e_fun = fun (x : Num) ->
    if x =? 0 then False
    else if x =? 1 then True
    else e_odd(x - 2)
e_odd = fix odd is e_fun
```

The proof:

$e_{odd}(\underline{3})$	(E-Fix)
$\equiv e_{fun}(\underline{3})$	(E-Ap)
$\equiv e_{odd}(\underline{1})$	(E-Fix)
$\equiv e_{fun}(\underline{1})$	(E-Ap)
$\equiv$ True	

Recursive Types

Motivations

- We've already seen these!
  - Lists
  - Trees
  - Syntax (trees)
- Again, we want recursive data types

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

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

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

Recursive types to the rescue!

- The type variable, a, is the self reference, and it stands for the recursive type itself in the recursive type's body, τ
  - `rec a is τ`
- BTree would be defined in ALFA as:

```
rec btree is Unit + Int x btree x btree
```

- `rec` is to *types* as `fix` is to *expressions*

Translating to ALFA

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

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

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

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

```
type BTree =
  rec BTree is 1 + (Num x BTree x
BTree)
```

```
type ALFExpr =
  rec A is Num + (A x A) + …
```

roll and unroll for BTree

$$\frac{\Gamma \vdash e_{\text{body}} : [(\text{rec } a \text{ is } \tau_{\text{body}})/a]\tau_{\text{body}}}{\Gamma \vdash \text{roll}(e_{\text{body}}) : \text{rec } a \text{ is } \tau_{\text{body}}} \quad \text{(T-Roll)}$$

$$\frac{\Gamma \vdash e : \text{rec } a \text{ is } \tau_{\text{body}}}{\Gamma \vdash \text{unroll}(e) : [(\text{rec } a \text{ is } \tau_{\text{body}})/a]\tau_{\text{body}}} \quad \text{(T-Unroll)}$$

```
BTree          = rec btree is Unit + Int x btree x btree
BTree_unrolled = Unit + Int x (rec btree is Unit + btree x btree) x (rec btree is Unit + btree x btree)

roll    : BTree_unrolled -> BTree
roll    : Unit + Int x (rec btree is Unit + btree x btree) x (rec btree is Unit + btree x btree)
          -> rec btree is Unit + Int x btree x btree
```

(introduction form)

```
unroll  : BTree          -> BTree_unrolled
unroll  : rec btree is Unit + Int x btree x btree
         -> Unit + Int x (rec btree is Unit + btree x btree) x (rec btree is Unit + btree x btree)
```

(elimination form)

roll and unroll for BTree

```
let leaf be (roll((L() : BTree_unrolled)) : BTree) in

let branch be fun (n : Int, l : BTree, r : BTree) ->
                 (roll((R(n, l, r) : BTree_unrolled)) : BTree) in

let height be fun (t : BTree) ->
    case (unroll(t) : BTree_unrolled) of
        L() -> 1 else
        R(_, l : BTree, r : BTree) -> max(height(l), height(r)) + 1
```

- Constructing BTree requires roll
- Consuming BTree requires unroll

Evaluation semantics

Q: What's the result of evaluating `unroll(roll(3))`?

A: 3. Rolling and then unrolling a value gives the value back.

$$\frac{e \Downarrow v}{\mathsf{roll}(e) \Downarrow \mathsf{roll}(v)} \text{ (Eval-Roll)} \qquad \frac{e \Downarrow \mathsf{roll}(v)}{\mathsf{unroll}(e) \Downarrow v} \text{ (Eval-Unroll)}$$

Type validity(?!)

Notice that we now might have *unbound type variables*(!)

`UhOh = rec a is Unit + b` (b is unbound!)

Solution: A type validity judgment

$$\frac{t \text{ valid} \in \Delta}{\Delta \vdash t \text{ valid}}\ (\text{TV-TVar})$$

$$\frac{\Delta, t \text{ valid} \vdash \tau_{body} \text{ valid}}{\Delta \vdash \text{rec } t \text{ is } \tau_{body} \text{ valid}}\ (\text{TV-Rec})$$

α-equivalence ("alpha-equivalence")

- Expressions e_1 and e_2 are α-equivalent if they are "basically equivalent up to bound variable names"
    - i.e. they have the *same binding structure*
- Same with types: `rec t is Unit + t` $\equiv_\alpha$ `rec nat is Unit + nat`

De Bruijn Indices

(solves problems)

Representing terms without variable names

With De Bruijn indices, you can write expressions and types without variable names:

```
fun x -> fun y -> fun z -> (x(z))(y(z))
```

becomes

```
fun · -> fun · -> fun · -> (2(0))(1(0))
```

Each variable occurrence is represented by a natural number that denotes the *number of binders in scope* between that occurrence and its binder

Recursive types with De Bruijn Indices

We can rewrite our recursive types from before:

```
Nat   = rec · is Unit + 0
List  = rec · is Unit + Int x 0
BTree = rec · is Unit + Int x 0 x 0
```

But why?

- α-equivalence is made trivial

  `rec t is Unit + t` and `rec nat is Unit + nat`

  become syntactically identical: `rec · is Unit + 0`

- Makes safe substitution easier
- Nice when using proof assistants

Some practice

- Convert the types to / from their De Bruijn representation

REC X IS (X + (REC Y IS (X + NUM)))

REC · IS (REC · IS (REC · IS (REC · IS ($\overleftarrow{0} + \overleftarrow{2} + \overleftarrow{3}$))))

Some practice

- Convert the types to / from their De Bruijn representation

REC X IS (X + (REC Y IS (X + NUM)))

REC · IS ($\overleftarrow{0}$ + (REC · IS ($\overleftarrow{1}$ + NUM)))

REC · IS (REC · IS (REC · IS (REC · IS ($\overleftarrow{0}$ + $\overleftarrow{2}$ + $\overleftarrow{3}$))))

REC P IS (REQ Q IS (REC R IS (REC T IS (T + Q + P))))