HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

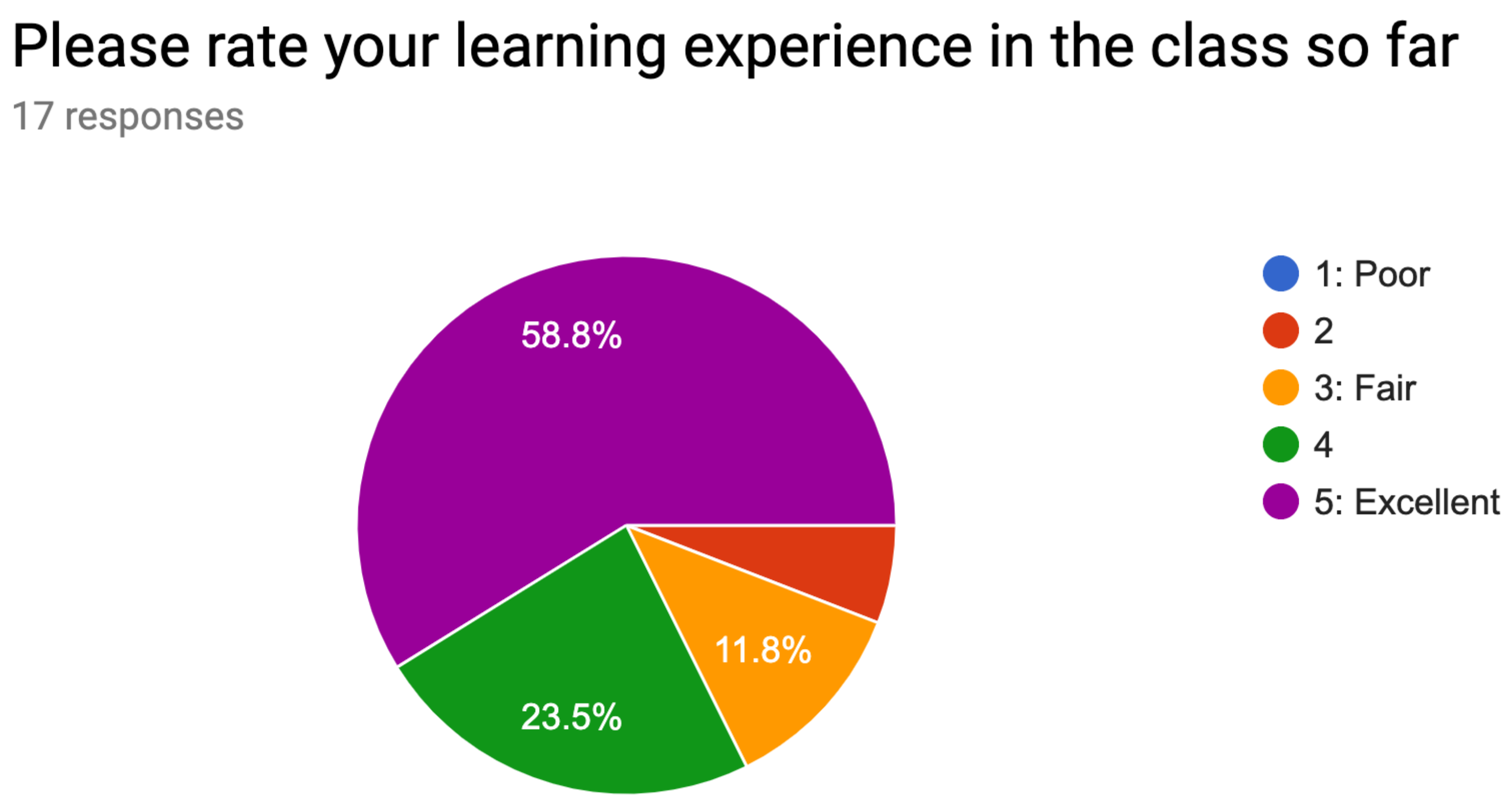
CS153: Compilers
Lecture 13:
Compiling functions

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

Mid-course Eval

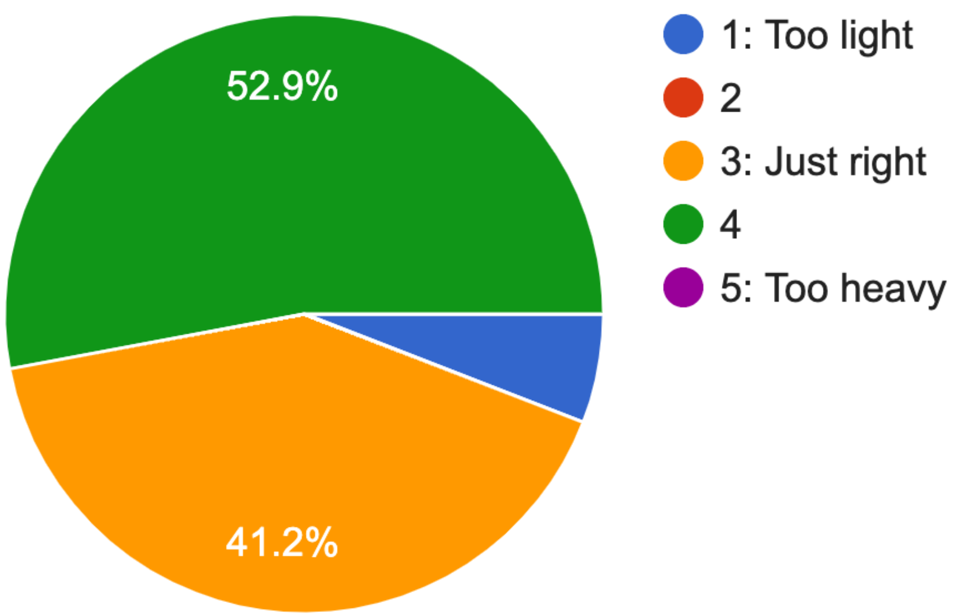


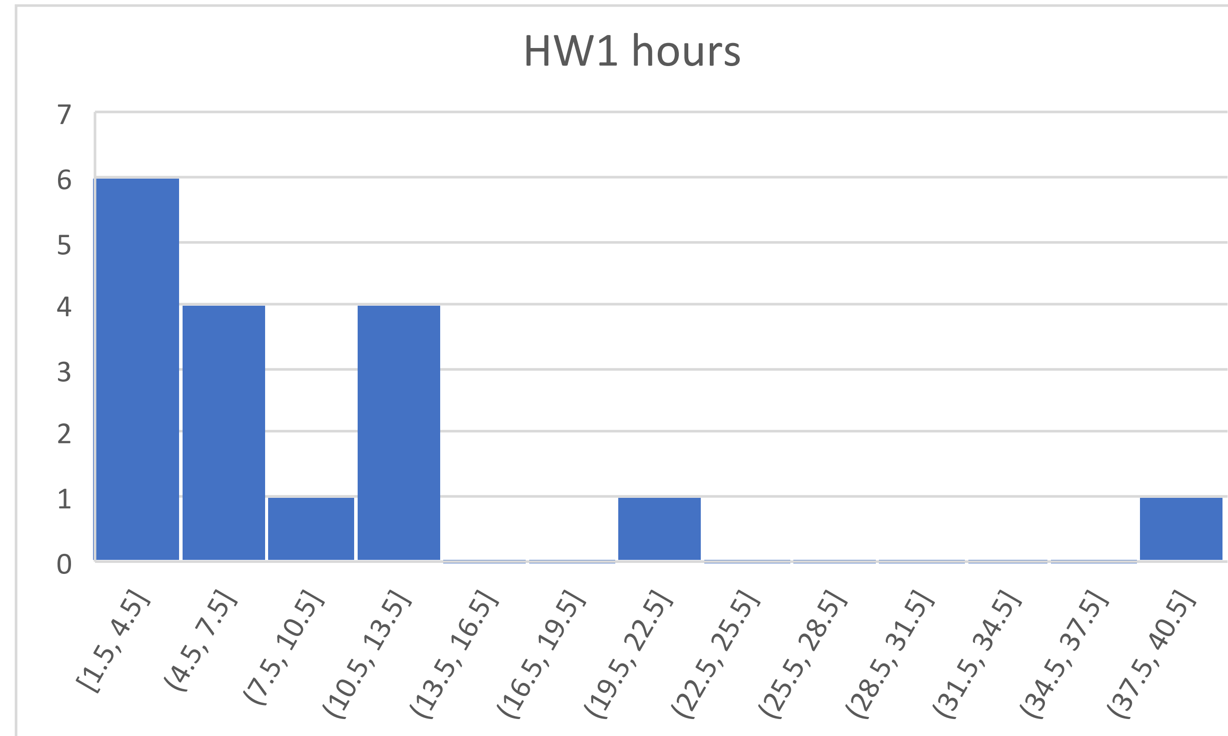
- Most effective:
  - Homeworks
  - Lectures
  - Piazza/OH
- Least effective:
  - Looking at code in class (too much, too little!)
  - Lecture material not relevant to current assignment
  - OH for extension

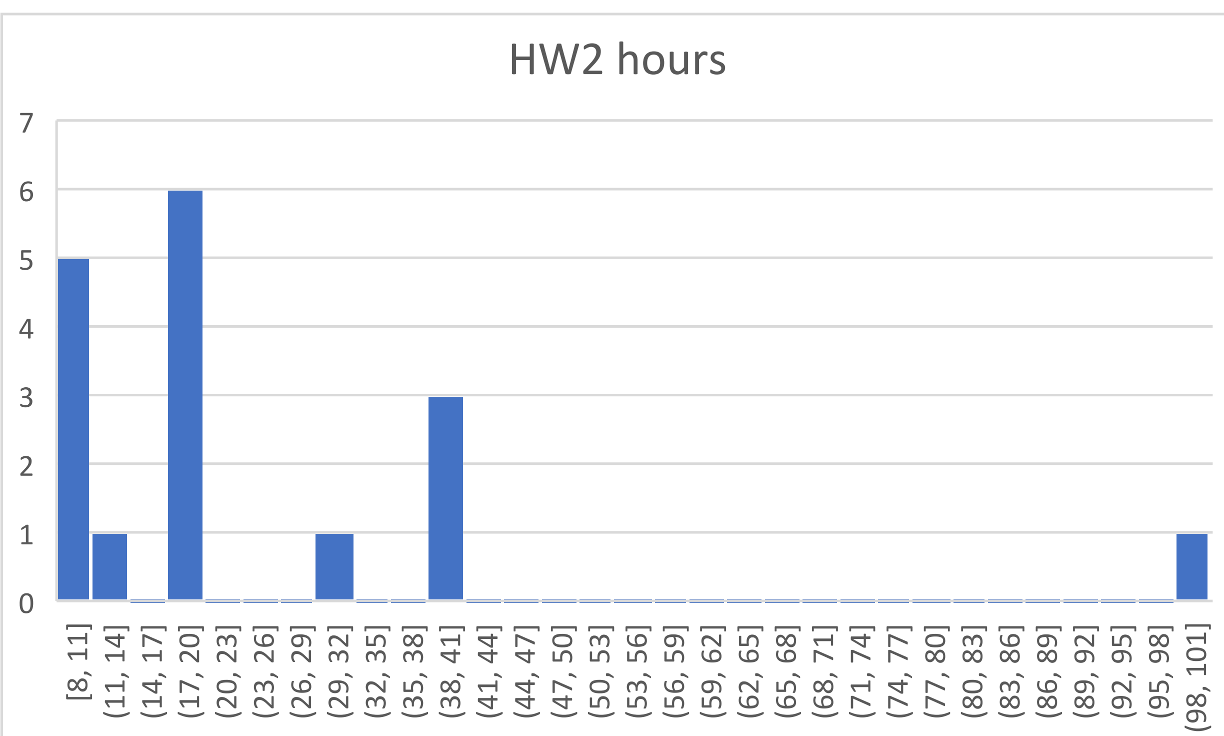
Mid-course Eval

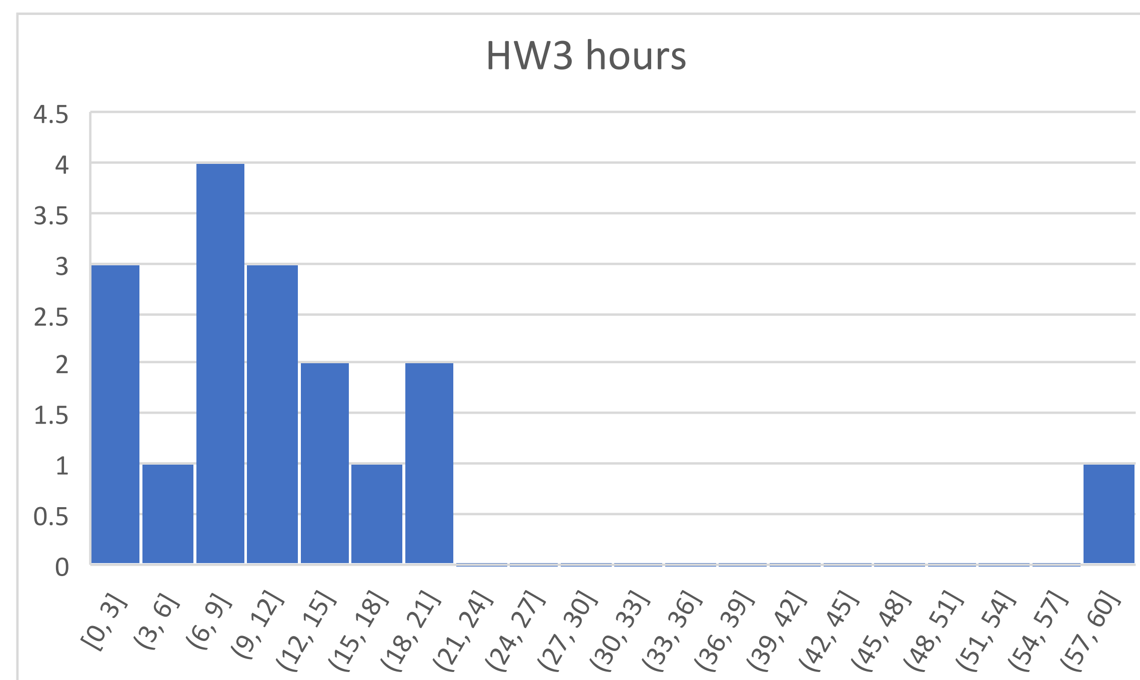
- Suggestions
  - Homework solutions
  - Idiomatic OCaml code
  - Go faster
  - “Stop using ocaml, it gets in the way of learning about compilers”, “This is not supposed to be an ocaml course, it's supposed to be a compilers course”
  - More type annotations in homework stub code
  - Long time to get OCaml set up
  - …

Workload


52.9%
41.2%
1: Too light
2
3: Just right
4
5: Too heavy


HW1 hours
0
1
2
3
4
5
6
7
[1.5, 4.5]
(4.5, 7.5]
(7.5, 10.5]
(10.5, 13.5]
(13.5, 16.5]
(16.5, 19.5]
(19.5, 22.5]
(22.5, 25.5]
(25.5, 28.5]
(28.5, 31.5]
(31.5, 34.5]
(34.5, 37.5]
(37.5, 40.5]


HW2 hours
0
1
2
3
4
5
6
7
[8, 11]
(11, 14]
(14, 17]
(17, 20]
(20, 23]
(23, 26]
(26, 29]
(29, 32]
(32, 35]
(35, 38]
(38, 41]
(41, 44]
(44, 47]
(47, 50]
(50, 53]
(53, 56]
(56, 59]
(59, 62]
(62, 65]
(65, 68]
(68, 71]
(71, 74]
(74, 77]
(77, 80]
(80, 83]
(83, 86]
(86, 89]
(89, 92]
(92, 95]
(95, 98]
(98, 101]


HW3 hours
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
[0, 3]
(3, 6]
(6, 9]
(9, 12]
(12, 15]
(15, 18]
(18, 21]
(21, 24]
(24, 27]
(27, 30]
(30, 33]
(33, 36]
(36, 39]
(39, 42]
(42, 45]
(45, 48]
(48, 51]
(51, 54]
(54, 57]
(57, 60]

Mid-course Eval: Actions

- Concrete actions course staff:
  - More type annotations in future HWs
  - Will release reference solutions
  - Lectures will be same pace or a bit faster (but will still have lots of time for questions)
- Concrete actions students:
  - Contact course staff re OH frequency/timing; we will try to adjust
  - Contact for additional info/feedback on graded HWs
  - Start HW early, reach out early and often for help
- Notes:
  - Implementation course: coding/coding style is important
  - Pedagogical decision to release HWs only after material is covered

Today

- Closure conversion
- Implementing environments and variables
  - DeBruijn indices
  - Nested environments vs flat environments

Closures

- Instead of doing substitution on nested functions when we reach the lambda, we can instead make a promise to finish the substitution if the nested function is ever applied
- Instead of

```
  | Lambda(x,e') -> Lambda(x,subst env e')
```

  we will have, in essence,

```
  | Lambda(x,e') -> Promise(env, Lambda(x, e'))
```

  - Called a **closure**
- Need to modify rule for application to expect environment

Closure-based Semantics

```
type value = Int_v of int
             | Closure_v of {env:env, body:var*exp}
and  env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
  | Int i -> Int_v i
  | Var x -> lookup env x
  | Lambda(x,e) -> Closure_v{env=env, body=(x,e)}
  | App(e1,e2) ->
      (match eval e1 env, eval e2 env with
       | Closure_v{env=cenv, body=(x,e')}, v ->
                 eval e' ((x,v)::cenv))
```

Inference rules

$$\frac{}{\Gamma \vdash i \Downarrow i}$$

$$\frac{\Gamma(x) = v}{\Gamma \vdash x \Downarrow v}$$

$$\frac{\Gamma \vdash e1 \Downarrow i1 \qquad \Gamma \vdash e2 \Downarrow i2 \qquad i = i1 + i2}{\Gamma \vdash e1 + e2 \Downarrow i}$$

$$\frac{}{\Gamma \vdash \text{fun } x \text{ -> } e \Downarrow (\Gamma, \text{fun } x \text{ -> } e)}$$

$$\frac{\Gamma \vdash e1 \Downarrow (\Gamma_c, \text{fun } x \text{ -> } e) \qquad \Gamma \vdash e2 \Downarrow v \qquad \Gamma_c[x \mapsto v] \vdash e \Downarrow w}{\Gamma \vdash e1\ e2 \Downarrow w}$$

So, How Do We Compile Closures?

- Represent function values (i.e., closures) as a pair of function pointer and environment
- Make all functions take environment as an additional argument
  - Access variables using environment

Closure conversion

- Can then move all function declarations to top level (i.e., no more nested functions!)

Lambda lifting

- E.g., `fun x -> (fun y -> y+x)` becomes, in C-like code:

```
closure *f1(env *env, int x) {
  env *e1 = extend(env,“x”,x);
  closure *c =
       malloc(sizeof(closure));
  c->env = e1; c->fn = &f2;
  return c;
}
```

```
int f2(env *env, int y) {
  env *e1 = extend(env,“y”,y);
  return lookup(e1, “y”)
           + lookup(e1, “x”);
}
```

Where Do Variables Live

- Variables used in outer function may be needed for nested function
  - e.g., variable **`x`** in example on previous slide
- So variables used by nested functions can't live on stack...
- Allocate record for all variables on heap
- This will be similar to objects (which we will see in a few lectures)
  - Object = struct for field values, plus pointer(s) to methods
  - Closure = environment plus pointer to code

Closure Conversion

- Converting function values into closures
  - Make all functions take explicit environment argument
  - Represent function values as pairs of environments and lambda terms
  - Access variables via environment
- E.g.,

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

becomes

```
fun env x ->
      let e’ = extend env “x” x in
      (e’, fun env y ->
            let e’ = extend env “y” y in
            (lookup e’ “y”)+(lookup e’ “x”))
```

Lambda Lifting

- After closure conversion, nested functions do not directly use variables from enclosing scope
- Can "lift" the lambda terms to top level functions!
- E.g.,

```
fun env x ->
          let e' = extend env "x" x in
          (e', fun env y ->
                 let e' = extend env "y" y in
                 (lookup e' "y")+(lookup e' "x"))
```

becomes

```
let f2 = fun env y ->
                let e' = extend env "y" y in
                (lookup e' "y")+(lookup e' "x")
fun env x ->
         let e' = extend env "x" x in
         (e', f2)
```

Lambda Lifting

- E.g., `fun env x ->`

```
              let e' = extend env "x" x in
              (e', fun env y ->
                    let e' = extend env "y" y in
                    (lookup e' "y")+(lookup e' "x"))
```

becomes

```
let f2 = fun env y ->
              let e' = extend env "y" y in
              (lookup e' "y")+(lookup e' "x")
fun env x ->
          let e' = extend env "x" x in
          (e', f2)
```

```
closure *f1(env *env, int x) {
  env *e1 = extend(env,"x",x);
  closure *c =
       malloc(sizeof(closure));
  c->env = e1; c->fn = &f2;
  return c;
}
```

```
int f2(env *env, int y) {
  env *e1 = extend(env,"y",y);
  return lookup(e1, "y")
          + lookup(e1, "x");
}
```

How Do We Compile Closures Efficiently?

- Don’t need to heap allocate all variables
  - Just the ones that “escape”, i.e., might be used by nested functions
- Implementation of environment and variables

DeBruijn Indices

- In our interpreter, we represented environments as lists of pairs of variables names and values
- Expensive string comparison when looking up variable! `lookup env x`

```
let rec lookup env x =
  match env with
  | ((y,v)::rest) ->
        if y = x then v else lookup rest
  | [] -> error "unbound variable"
```

- Instead of using strings to represent variables, we can use natural numbers
    - Number indicates lexical depth of variable

DeBruijn Indices

```
type exp = Int of int | Var of int
         | Lambda of exp | App of exp*exp
```

- Original program

```
fun x -> fun y -> fun z -> x + y + z
```

- Conceptually, can rename program variables

```
fun x2 -> fun x1 -> fun x0 -> x2 + x1 + x0
```

- Don't bother with variable names at all!

```
fun -> fun -> fun -> Var 2 + Var 1 + Var 0
```

  - Number of variable indicates lexical depth, 0 is innermost binder

Converting to DeBruijn Indices

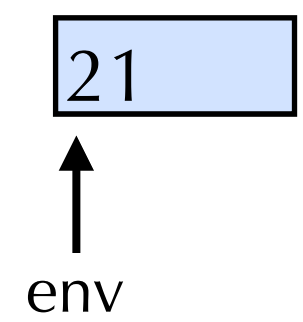
```
type exp = Int of int | Var of int
         | Lambda of exp | App of exp*exp

let rec cvt (e:exp) (env:var->int): D.exp =
  match e with
  | Int i -> D.Int i
  | Var x -> D.Var (env x)
  | App(e1,e2) ->
      D.App(cvt e1 env,cvt e2 env)
  | Lambda(x,e) =>
      let new_env(y) =
            if y = x then 0 else (env y)+1
      in
         Lambda(cvt e new_env)
```

New Interpreter

```
type value = Int_v of int
           | Closure_v of {env:env, body:exp}
and  env = value list

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var x -> List.nth env x
   | Lambda e -> Closure_v{env=env, body=e}
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
        | Closure_v{env=cenv, body=(x,e')}, v ->
                   eval e' v::cenv)
```

Representing Environments

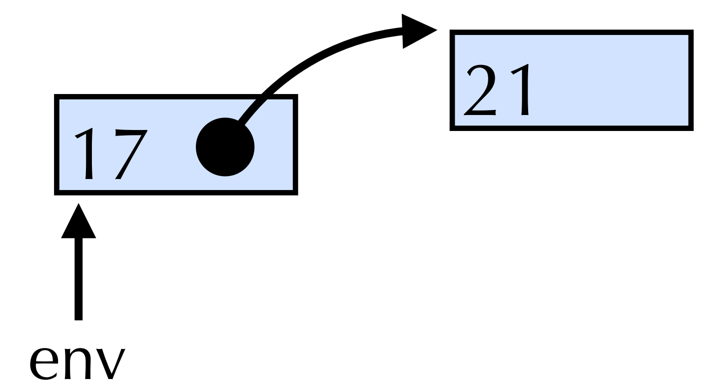
```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



- Linked list (nested environments)

Representing Environments

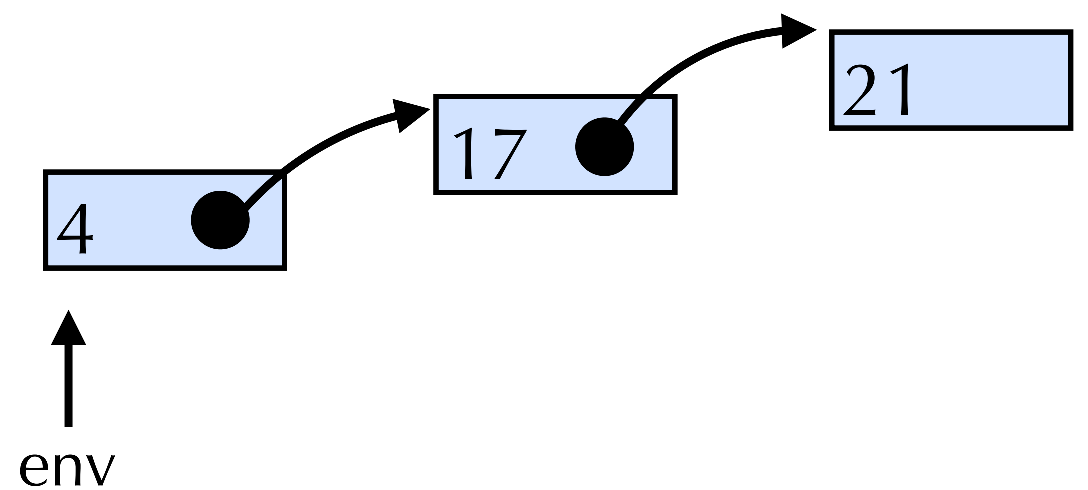
```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



- Linked list (nested environments)

Representing Environments

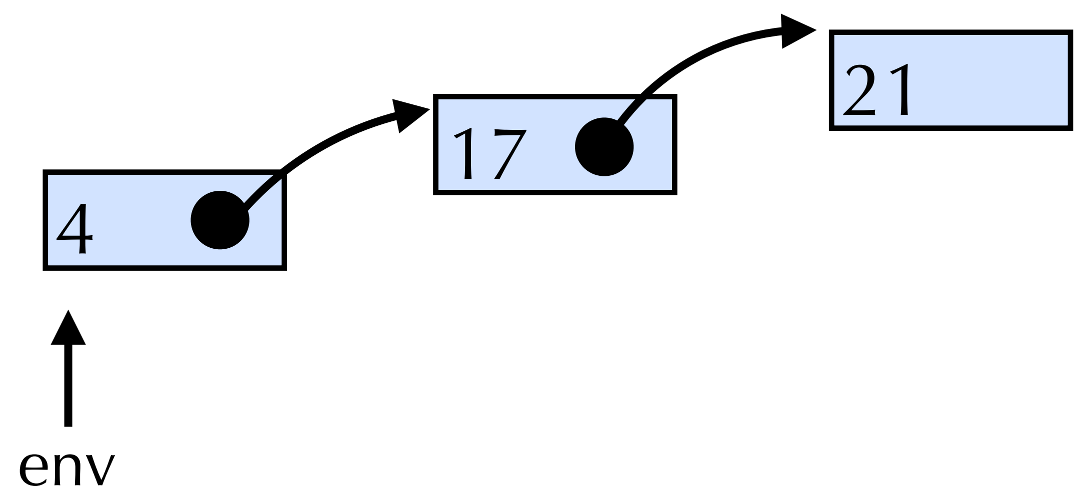
```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



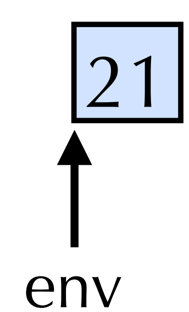
- Linked list (nested environments)

Representing Environments

```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```

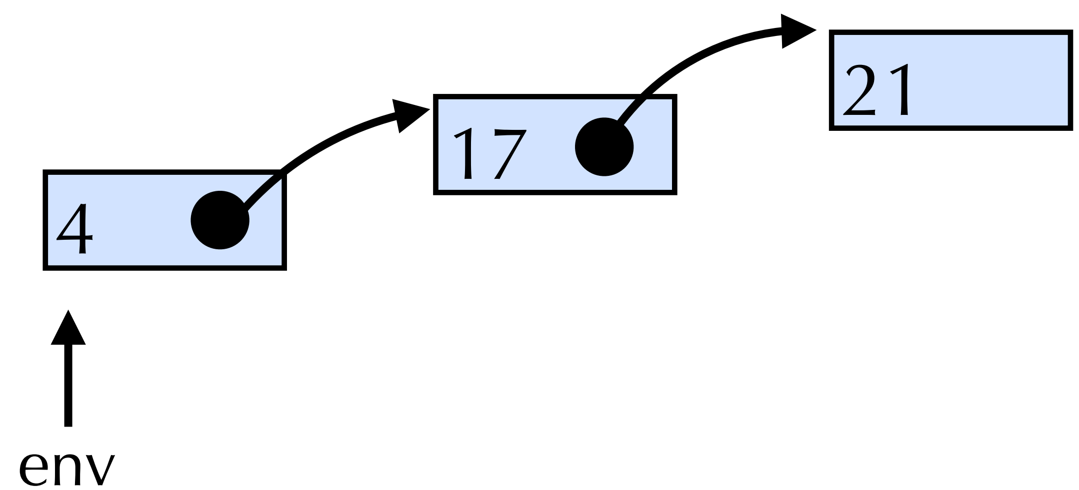


- Linked list (nested environments)
- Array (flat environment)

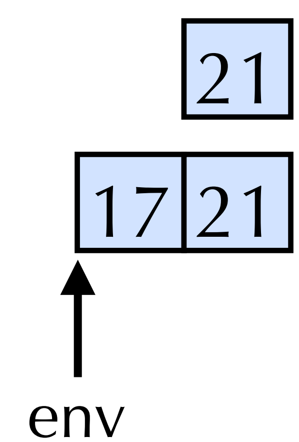


Representing Environments

```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```

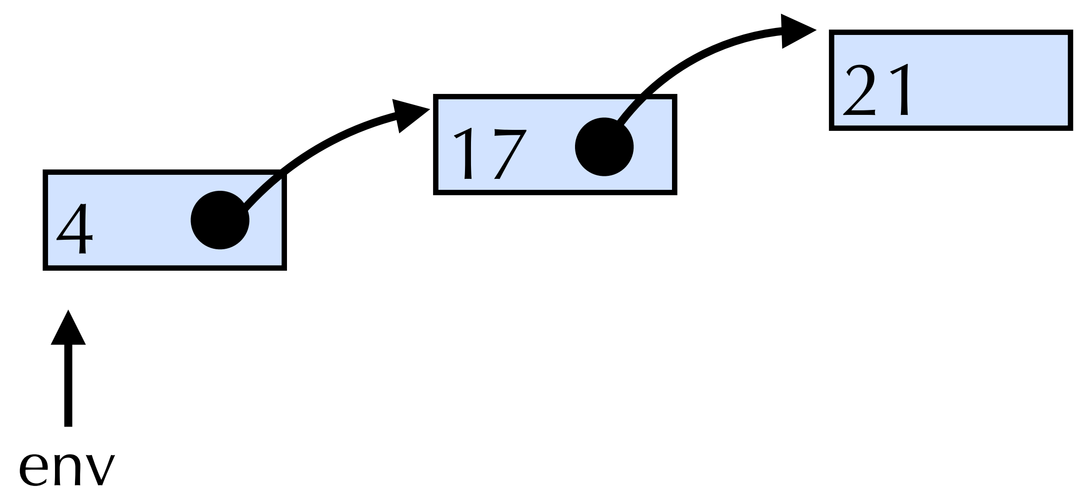


- Linked list (nested environments)
- Array (flat environment)

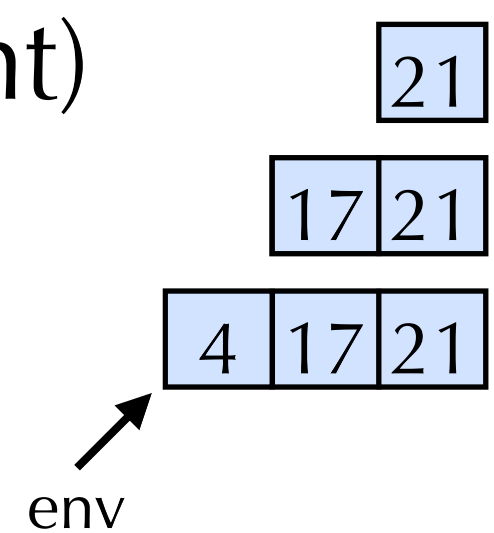


Representing Environments

```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



- Linked list (nested environments)
- Array (flat environment)

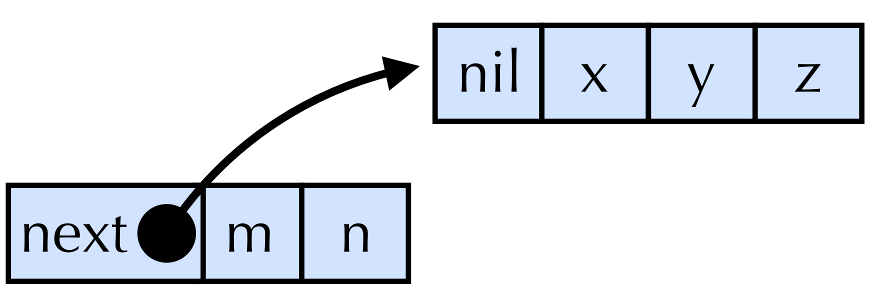


Multiple Arguments

- Can extend DeBruijn indices to allow multiple arguments

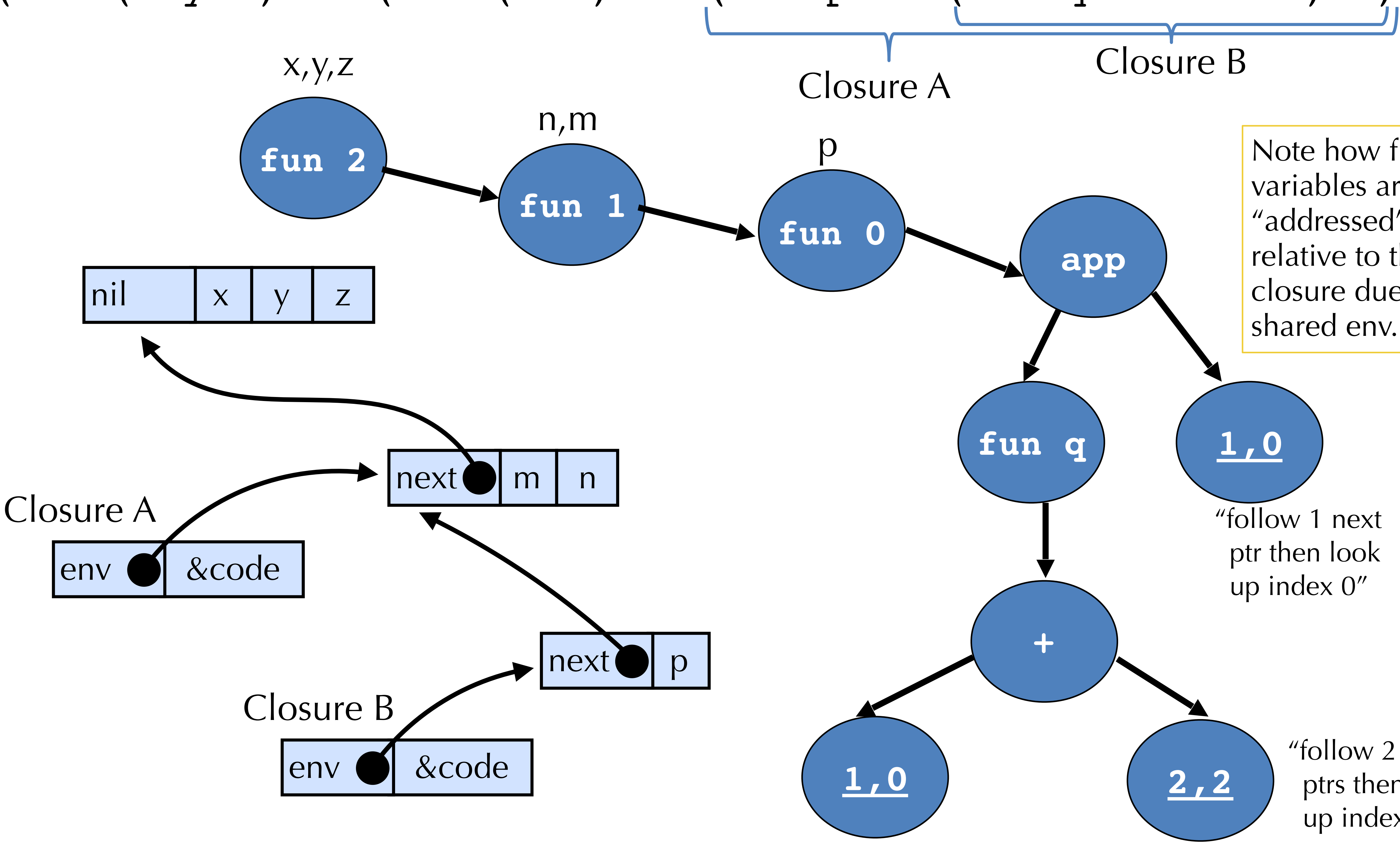
```
fun x y z -> fun m n -> x + z + n

fun -> fun-> Var(1,0) + Var(1,2) + Var(0,1)
```

- Nested environments might then be



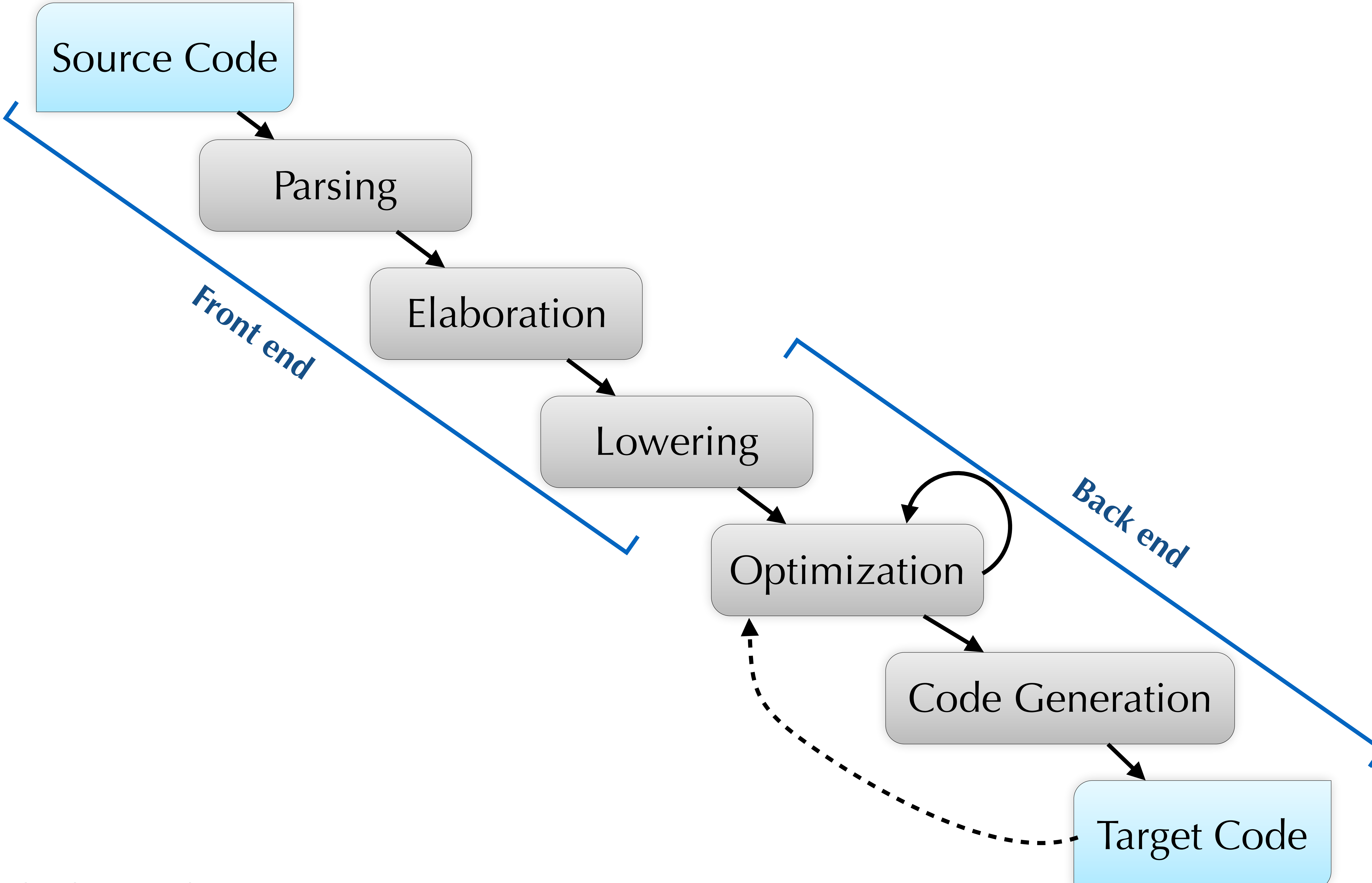
Array-based Closures with N-ary Functions

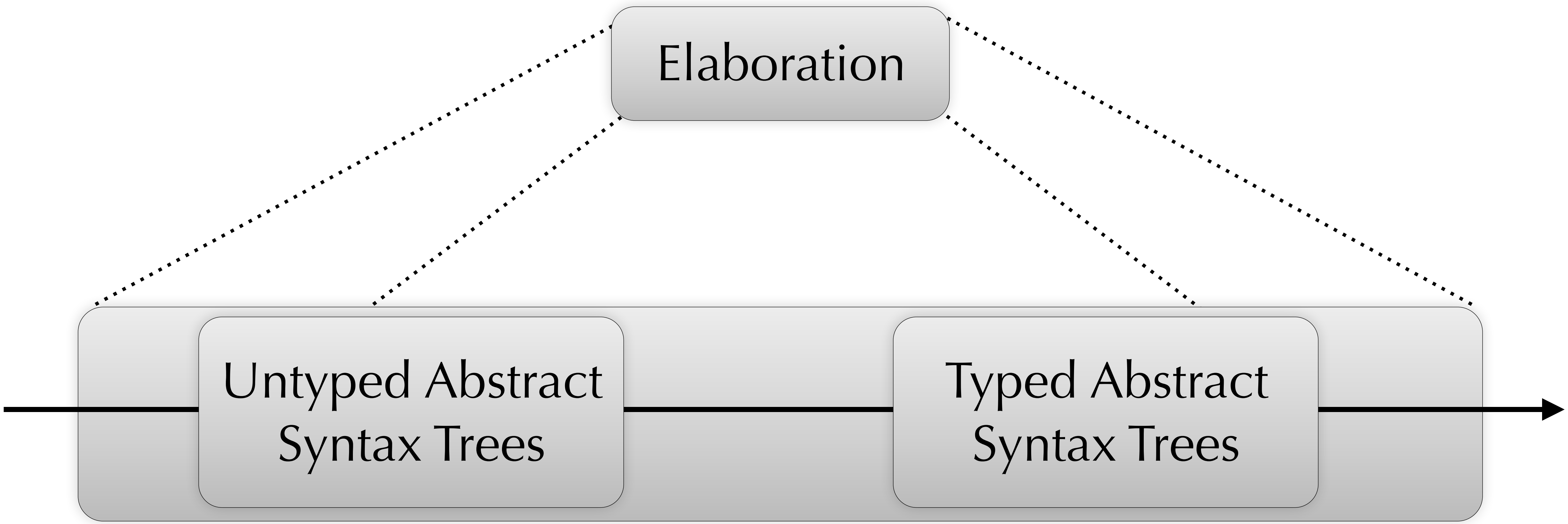
```
(fun (x y z) -> (fun (m n) -> (fun p -> (fun q -> m + z) x)
```



Note how free variables are "addressed" relative to the closure due to shared env.

Basic Architecture




Elaboration
Untyped Abstract Syntax Trees
Typed Abstract Syntax Trees

Undefined Programs

- After parsing, we have AST
- We can interpret AST, or compile it and execute
- But: not all programs are well defined
  - E.g., `3/0`, `“hello” - 7`, `42(19),` using a variable that isn’t in scope, ...
- **Types** allow us to rule out many of these undefined behaviors
  - Types can be thought of as an approximation of a computation
  - E.g., if expression `e` has type `int`, then it means that `e` will evaluate to some integer value
  - E.g., we can ensure we never treat an integer value as if it were a function

Type Soundness

- Key idea: a well-typed program when executed does not attempt any undefined operation
- Make a model of the source language
  - i.e., an interpreter, or other semantics
  - This tells us which operations are partial
  - Partiality is different for different languages
    - E.g., `"Hi" + " world"` and `"na"*16` may be meaningful in some languages
- Construct a function to check types: `tc : AST -> bool`
  - AST includes types (or type annotations)
  - If `tc e` returns true, then interpreting `e` will not result in an undefined operation
- Prove that `tc` is correct

Simple Language

```
type tipe =
    Int_t
  | Arrow_t of tipe*tipe
  | Pair_t of tipe*tipe

type exp =
    Var of var | Int of int
  | Plus_i of exp*exp
  | Lambda of var * tipe * exp
  | App of exp*exp
  | Pair of exp * exp
  | Fst of exp | Snd of exp
```

Note: function arguments have type annotation

Interpreter

```
let rec interp (env:var->value)(e:exp) =
  match e with
   | Var x -> env x
   | Int i -> Int_v i
   | Plus_i(e1,e2) ->
      (match interp env e1, interp env e2 of
         | Int_v i, Int_v j -> Int_v(i+j)
         | _,_ -> failwith “Bad operands!”)
   | Lambda(x,t,e) -> Closure_v{env=env,code=(x,e)}
   | App(e1,e2) ->
     (match (interp env e1, interp env e2) with
         | Closure_v{env=cenv,code=(x,e)},v ->
               interp (extend cenv x v) e
         | _,_ -> failwith “Bad operands!”)
```

Type Checker

```
let rec tc (env:var->tipe) (e:exp) =
  match e with
   | Var x -> env x
   | Int _ -> Int_t
   | Plus_i(e1,e2) ->
      (match tc env e1, tc env e with
         | Int_t, Int_t -> Int_t
         | _,_ -> failwith "...")
   | Lambda(x,t,e) -> Arrow_t(t,tc (extend env x t) e)
   | App(e1,e2) ->
     (match (tc env e1, tc env e2) with
         | Arrow_t(t1,t2), t ->
             if (t1 != t) then failwith "..." else t2
         | _,_ -> failwith "...")
```

Notes

- Type checker is almost like an **approximation** of the interpreter!
  - But interpreter evaluates function body only when function applied
  - Type checker always checks body of function
- We needed to assume the input of a function had some type t_1, and reflect this in type of function ($t_1 \texttt{->} t_2$)
- At call site ($e_1\ e_2$), we don't know what closure e_1 will evaluate to, but can calculate type of e_1 and check that e_2 has type of argument

Growing the Language

- Adding booleans...

```
type tipe = ... | Bool_t

type exp = ... | True | False | If of exp*exp*exp

let rec interp env e = ...
| True -> True_v
| False -> False_v
| If(e1,e2,e3) -> (match interp env e1 with
                     True_v -> interp env e2
                   | False_v -> interp env e3
                   | _ -> failwith “...”)
```

Type Checking

```
let rec tc (env:var->tipe) (e:exp) =
  match e with
  ...
  | True -> Bool_t
  | False -> Bool_t
  | If(e1,e2,e3) ->
   (let (t1,t2,t3) = (tc env e1,tc env e2,tc env e3)
    in
       match t1 with
       | Bool_t ->
           if (t2 != t3) then error() else t2
       | _ -> failwith "...")
```

Type Safety

- "Well typed programs do not go wrong."
  – Robin Milner, 1978
- Note: this is a **very** strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)
- Depending on language, will not rule out **all** possible undefined behavior
  - E.g., 3/0, *NULL, ...
  - More sophisticated type systems can rule out more kinds of possible runtime errors

Judgements and Inference Rules

- We saw type checking algorithm in code
- Can express type-checking rules compactly and clearly using a **type judgment** and **inference rules**

Type Judgments

- In the judgment: $E \vdash e : t$
  - E is a typing environment or a type context
  - E maps variables to types. It is just a set of bindings of the form: x1 : t1, x2 : t2, …, xn : tn
- If $E \vdash e : t$ then expression e has type t under typing environment E
  - $E \vdash e : t$ can be thought of as a set or relation
- For example:

  $$x : \text{int}, b : \text{bool} \vdash \text{if (b) 3 else x} : \text{int}$$

- What do we need to know to decide whether "if (b) 3 else x" has type int in the environment x : int, b : bool?
  - b must be a bool i.e. $x : \text{int}, b : \text{bool} \vdash b : \text{bool}$
  - 3 must be an int i.e. $x : \text{int}, b : \text{bool} \vdash 3 : \text{int}$
  - x must be an int i.e. $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

Why Inference Rules?

- Compact, precise way of specifying language properties.
  - E.g. ~20 pages for full Java vs. 100's of pages of prose Java Language Spec.
- Inference rules correspond closely to the recursive AST traversal that implements them
- Type checking (and type inference) is nothing more than attempting to prove a different judgment ($E \vdash e : t$) by searching backwards through the rules.
- Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment ($G \vdash src \Rightarrow target$)
  - Moreover, the compilation rules are very similar in structure to the typechecking rules
- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CS152 if you're interested in type systems!

Inference Rules

- For Oat, we will split environment E into global variables G and local variables L
- Judgment $\texttt{G;L} \vdash e : t$ "expression e is well typed and has type t"
- Judgment $\texttt{G;L} \vdash s$ "statement s is well formed"

Premises: $\texttt{G;L} \vdash e : \text{bool} \qquad \texttt{G;L} \vdash s_1 \qquad \texttt{G;L} \vdash s_2$

Conclusion: $\texttt{G;L} \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2$

$$\frac{\texttt{G;L} \vdash e : \text{bool} \qquad \texttt{G;L} \vdash s_1 \qquad \texttt{G;L} \vdash s_2}{\texttt{G;L} \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

- Equivalently: For any environment G; L, expression e, and statements s_1, s_2.

$$\texttt{G;L} \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2$$

holds if $\texttt{G;L} \vdash e : \text{bool}$ and $\texttt{G;L} \vdash s_1$ and $\texttt{G;L} \vdash s_2$ all hold.

- This rule can be used for *any* substitution of the syntactic metavariables G, L e, s_1 and s_2

Simply-typed Lambda Calculus

INT

$$\frac{}{E \vdash i : \text{int}}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \;:\; T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

- Note how these rules correspond to the code.

Type Checking Derivations

- A **derivation** or **proof tree** is a tree where nodes are instantiations of inference rules and edges connect a premise to a conclusion
- Leaves of the tree are **axioms** (i.e. rules with no premises)
  - E.g., the INT rule is an axiom
- Goal of the typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:

$$\vdash \text{(fun (x:int) -> x + 3) 5} \;:\; \text{int}$$

Example Derivation Tree

$$\text{APP}\;\dfrac{\text{FUN}\;\dfrac{\text{ADD}\;\dfrac{\text{VAR}\;\dfrac{x : \text{int} \in x : \text{int}}{x : \text{int} \vdash x : \text{int}} \qquad \text{INT}\;\dfrac{}{x : \text{int} \vdash 3 : \text{int}}}{x : \text{int} \vdash x + 3 : \text{int}}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3) : \text{int -> int}} \qquad \text{INT}\;\dfrac{}{\vdash 5 : \text{int}}}{\vdash (\text{fun } (x{:}\text{int}) \text{ -> } x + 3)\ 5 : \text{int}}$$

- Note: the OCaml function typecheck verifies the existence of this tree. The structure of the recursive calls when running `tc` is same shape as this tree!
- Note that $x : \text{int} \in E$ is implemented by the function `env`

Type Safety Revisited

Theorem: (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$.

Arrays

- Array constructs are not hard
- First: add a new type constructor: T[]

NEW

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : T}{E \vdash \text{new } T[e_1](e_2) \ : T[]}$$

e_1 is the size of the newly allocated array. e_2 initializes the elements of the array.

INDEX

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int}}{E \vdash e_1[e_2] \ : T}$$

Note: These rules don't ensure that the array index is in bounds – that should be checked *dynamically*.

UPDATE

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int} \qquad E \vdash e_3 : T}{E \vdash e_1[e_2] = e_3 \text{ ok}}$$

Tuples

- ML-style tuples with statically known number of products:
- First: add a new type constructor: $T_1 * \ldots * T_n$

TUPLE

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ

$$\frac{E \vdash e : T_1 * \ldots * T_n \quad 1 \leq i \leq n}{E \vdash \#i\ e \ : \ T_i}$$

References

- ML-style references (note that ML uses only expressions)
- First, add a new type constructor: T ref

REF

$$\frac{E \vdash e : T}{E \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{E \vdash !e \ : T}$$

ASSIGN

$$\frac{E \vdash e_1 : T \text{ ref} \qquad E \vdash e_2 : T}{E \vdash e_1 := e_2 \ : \text{unit}}$$

Note the similarity with the rules for arrays…

Oat Type Checking

- For HW5 we will add typechecking to Oat
  - And some other features
- XXX typing rules for Oat
- Example derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0 ; \cdot ; \texttt{int} \vdash \texttt{var}\ x_1 = 0 ; \texttt{var}\ x_2 = x_1 + x_1 ; x_1 = x_1 - x_2 ; \texttt{return}\ x_1 ; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\ [\text{STMTS}]}{\vdash \texttt{var}\ x_1 = 0 ; \texttt{var}\ x_2 = x_1 + x_1 ; x_1 = x_1 - x_2 ; \texttt{return}\ x_1 ;}\ [\text{PROG}]$$

Example Derivation

$$\mathcal{D}_1 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\;[\textsc{INT}]}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\;[\textsc{CONST}]}{G_0\,;\cdot \vdash \mathtt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \mathtt{int}}\;[\textsc{DECL}]}{G_0\,;\,\cdot\,;\mathtt{int} \vdash \mathtt{var}\ x_1 = 0;\ \Rightarrow \cdot, x_1 : \mathtt{int}}\;[\textsc{SDECL}]$$

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : (\mathtt{int}, \mathtt{int}) \to \mathtt{int}}\;[\textsc{ADD}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\;[\textsc{VAR}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\;[\textsc{VAR}]}{G_0\,;\cdot, x_1 : \mathtt{int} \vdash x_1 + x_1 : \mathtt{int}}\;[\textsc{BOP}]}{G_0\,;\cdot, x_1 : \mathtt{int}; \mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1;\ \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\;[\textsc{DECL}]}{G_0\,;\cdot, x_1 : \mathtt{int}; \mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1;\ \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\;[\textsc{SDECL}]$$

Example Derivation

$$x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \;;$$

$$\mathcal{D}_3 \quad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{-} : (\texttt{int}, \texttt{int}) \to \texttt{int}}\;[\textsc{Add}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}\;[\textsc{Var}] \quad \dfrac{x_2 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_2 : \texttt{int}}\;[\textsc{Var}]}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}\;[\textsc{Bop}]}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\;[\textsc{Assn}]$$

$$\mathcal{D}_4 \quad = \quad \dfrac{\dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}\;[\textsc{Var}]}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash \texttt{return}\ x_1 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\;[\textsc{Ret}]$$

Type Safety For General Languages

Theorem: (Type Safety)

If $\vdash P : t$ is a well-typed program, then either:
(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

- Well-defined termination could include:
  - halting with a return value
  - raising an exception
- Type safety rules out undefined behaviors:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language
- What is "defined" depends on the language semantics…

Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:

$$C \vdash e : t$$

- How do we interpret this information in the target language?

$$[\![C \vdash e : t]\!] = \quad ?$$

- $[\![C]\!]$ translates contexts
- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand
- INVARIANT: if $[\![C \vdash e : t]\!]$ = ty, operand , stream
  then the type (at the target level) of the operand is ty=$[\![t]\!]$

Example

- $C \vdash 37 + 5 : \text{int}$ what is $[\![C \vdash 37 + 5 : \text{int}]\!]$?

$[\![\vdash 37 : \text{int}]\!] =$ `(i64, Const 37, [])`

$[\![C \vdash 37 : \text{int}]\!] =$ `(i64, Const 37, [])`

$[\![\vdash 5 : \text{int}]\!] =$ `(i64, Const 5, [])`

$[\![C \vdash 5 : \text{int}]\!] =$ `(i64, Const  5, [])`

$[\![C \vdash 37 + 5 : \text{int}]\!] =$ `(i64, %tmp, [%tmp = add i64 (Const 37) (Const 5)])`

What about the Context?

- What is ⟦C⟧?
- Source level C has bindings like: x:int, y:bool
  - We think of it as a finite map from identifiers to types
- What is the interpretation of C at the target level?
- ⟦C⟧ maps source identifiers, "x" to source types and ⟦x⟧
- What is the interpretation of a variable ⟦x⟧ at the target level?
  - How are the variables used in the type system?

$$\frac{x : t \in L}{G ; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

$$\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L \vdash x = exp ; \Rightarrow L}\ \text{TYP\_ASSN}$$

as addresses
(which can be assigned)

Interpretation of Contexts

- ⟦C⟧ = a map from source identifiers to types and target identifiers

- INVARIANT:

  x:t ∈ C means that

  (1) lookup ⟦C⟧ x = (t, `%id_x`)

  (2) the (target) type of `%id_x` is ⟦t⟧* (a pointer to ⟦t⟧)

Interpretation of Variables

- Establish invariant for expressions:

$$\left[\!\left[\frac{x{:}t \in L}{G;L \vdash x : t}\ \text{TYP\_VAR} \right]\!\right] = $$

as expressions
(which denote values)

```
(%tmp, [%tmp = load i64* %id_x])
```

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

- What about statements?

$$\left[\!\left[\frac{x{:}t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp; \Rightarrow L}\ \text{TYP\_ASSN} \right]\!\right] = \text{stream} \ @$$

as addresses
(which can be assigned)

```
[store ⟦t⟧ opn, ⟦t⟧* %id_x]
```

where (t, `%id_x`) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, `opn`, stream)

Other Judgments?

- Statement:
  ⟦C; rt ⊢ stmt ⇒ C′⟧ = ⟦C′⟧ , stream

- Declaration:
  ⟦G;L ⊢ t x = exp ⇒ G;L,x:t ⟧ = ⟦G;L,x:t⟧, stream

  INVARIANT: stream is of the form:

```
stream' @
[ %id_x = alloca ⟦t⟧;
store ⟦t⟧ opn, ⟦t⟧* %id_x ]
```

  and ⟦G;L ⊢ exp : t ⟧ = (⟦t⟧, `opn`, stream′)

- Rest follow similarly

Compiling Control

Translating while

- Consider translating "`while(e) s`":
  - Test the conditional, if true jump to the body, else jump to the label after the body.

```
⟦C;rt ⊢ while(e) s ⇒ C'⟧ = ⟦C'⟧,
lpre:
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
    ⟦C;rt ⊢ s ⇒ C'⟧

    br %lpre
lpost:
```

- Note: writing `opn = ⟦C ⊢ e : bool⟧` is pun
  - translating ⟦C ⊢ e : bool⟧ generates *code* that puts the result into `opn`
  - In this notation there is implicit collection of the code

Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge

⟦C;rt ... =

⟦C′⟧

```
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
     ⟦C;rt ⊢ s1 ⇒ C′⟧

     br %merge
else:
    ⟦C; rt s2 ⇒ C′⟧

     br %merge
merge:
```

Connecting this to Code

- Instruction streams:
  - Must include labels, terminators, and "hoisted" global constants
- Must post-process the stream into a control-flow-graph
- See frontend.ml from HW4