Types, conclusion

Read: Scott, Chapters 7.1-7.2 and 8

Announcements

- Quiz 8

- Welcome back!
- Check your Rainbow grades
  - Exam 1-2, Quiz 1-7, HW 1-5
- HW 6 due Wednesday at midnight
- HW 7 out

Quiz 8 Q1 to Q3

Combinators:

$tru = \lambda x. \lambda y. x$

$pair = \lambda f. \lambda s. \lambda b. b\ f\ s$

$fst = \lambda p. p\ tru$

Rules:

1) A combinator is a Lambda abstraction with no free variables. Top-level term is Abstraction.
2) When replacing abbreviation (e.g. pair) with corresponding λ-expression, always parenthesize!
3) Keep abbreviation as long as possible, until it is in some head position and we need to expand to λ-expression.

Q1: $tru\ v\ w = ((tru\ v)\ w) =$

$\underline{(\lambda x. \lambda y. x)\ v}\ w \rightarrow_\beta (\lambda y. v)\ w \rightarrow_\beta \boxed{v}$

Q2: $pair\ v\ w =$

$\underline{(\lambda f. \lambda s. \lambda b. b\ f\ s)\ v}\ w \rightarrow_\beta \underline{(\lambda s. \lambda b. b\ v\ s)\ w} \rightarrow \boxed{\lambda b. b\ v\ w}$

Q3: $fst\ (pair\ v\ w) \rightarrow^* fst\ (\lambda b. b\ v\ w) =$

$\underline{(\lambda p. p\ tru)\ (\lambda b. b\ v\ w)} \rightarrow_\beta$

$\underline{(\lambda b. b\ v\ w)\ tru} \rightarrow_\beta tru\ v\ w \rightarrow^* \boxed{v}$

(Q1 reduction)

Lecture Outline

- Types (last time)
- Type systems (last time)
  - Type checking
  - Type safety
- Type equivalence (last time)
- Types in C

- Primitive types
- Composite types

Type Equivalence

- Two ways of defining type equivalence
  - Structural equivalence: based on "shape"
    - Roughly, two types are the same if they consists of the same components, put together in the same way
  - Name equivalence: based on lexical occurrence of the type definition
    - Strict name equivalence: aliased types are distinct
    - Loose name equivalence: aliased types are same

```
T1 x; …
T2 y;
x = y; ✓  ✗
```

Exercise: Structural Equivalence

```
type cell = … // record type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```

p,q,r,s,t,u

Exercise: Loose Name Equivalence

```
type cell = … // record type
                        ANON1
type alink = pointer to cell
type blink = alink
                  ANON2
p,q : pointer to cell
r : alink
s : blink
                ANON3
t : pointer to cell
u : alink
```

p,q

r,s,u

t

Exercise: Strict Name Equivalence

```
type cell = ... // record type
```

ANON1

```
type alink = pointer to cell
type blink = alink
```

ANON2

```
p,q : pointer to cell
r : alink
s : blink
```

ANON3

```
t : pointer to cell
u : alink
```

p,q

r,u

s

t

Example: Type Equivalence in C

- First, in the Algol family, field names are part of the record/struct constructed type. E.g., the record types below are NOT even structurally equivalent

```
type A = record
 x,y : real
end;
type B = record
 z,w : real
end;
```

Type Equivalence in C

- Compiler assigns internal (compiler-generated) names to anonymous types

This **struct** is of type anon1.

```
struct RecA           typedef struct      struct
{ char x;             { char x;           { char x;
  int y;                int y;              int y;
} a;                  } RecB;             } c;

RecB b;
```

Each one is of distinct type.

What variables are of **equivalent type** according to the rules in C?

a b c

Type Equivalence in C

- C uses structural equivalence for everything, except unions and structs, for which it uses loose name equivalence

```
struct A
{ char x;
  int y;
}

struct B
{ char x;
  int y;
}

typedef struct A C;
typedef C *P;
typedef struct B *Q;
typedef struct A *R;
typedef int Age;
typedef int (*F) (int);
typedef Age (*G) (Age);
```

struct B

struct A, C

P, R

Q

int, Age

F, G

Type Equivalence in C

```
struct B { char x; int y; };
typedef struct B A;
struct { A a; A *next; } aa;
struct { struct B a; struct B *next; } bb;
struct { struct B a; struct B *next; } cc;

A a;
struct B b;

a = b;
aa = bb;
bb = cc;
```

Which of the above assignments pass the type checker?

Question

- Structural equivalence for record types is considered a bad idea. Can you think of a reason why? LOOSE NAME IS A BETTER CHOICE!

Lecture Outline

- Types
- Type systems
  - Type checking
  - Type safety
- Type equivalence
- Types in C

- Primitive types
- Composite types

Pointers and Arrays in C

- Pointers and arrays are interoperable:

```
int n;
int *a;
int b[10];
```

```
1.  a = b;
2.  n = a[3];
3.  n = *(a+3);
4.  n = b[3];
5.  n = *(b+3);
```

Declarations in C

- What is the meaning of the following declaration in C? Draw the type trees.

```
1. int *a[n]
2. int (*a)[n]
3. int (*f)(int)
```

1. a
t: array
t1: pointer
t2: int

2. a
t: pointer
t1: array
t2: int

3. f
t: pointer
→
(int) int

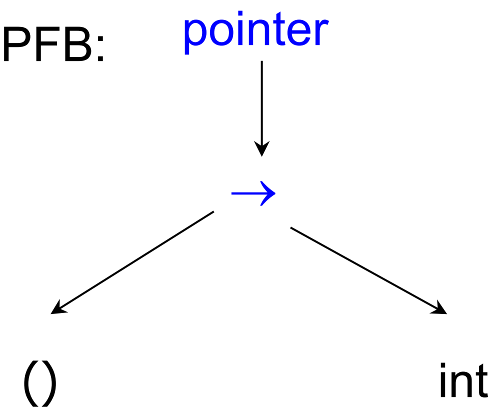
Declarations in C

```
typedef int (*PFB)();                  // Type variable PFB: what type?
struct parse_table {                   // Type struct parse_table: what type?
    char *name;
    PFB func; };
int func1() { ... }                    // Function func1: what type?
int func2() { ... }

struct parse_table table[] = {         // Variable table: what type?
    {"name1", &func1},
    {"name2", &func2}
};
PFB find_p_func(char *s) {             // Function find_p_func: what type?
    for (i=0; i<num_func; i++)
        if (strcmp(table[i].name,s)==0) return table[i].func;
    return NULL; }
int main(int argc,char *argv[]) {
    ... }
```

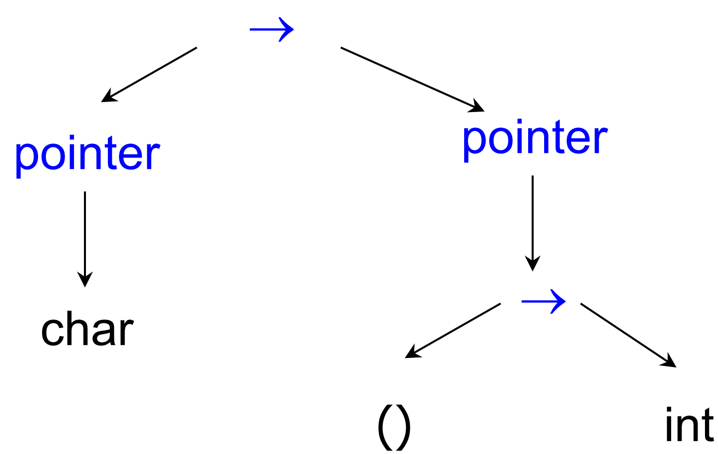
Declarations in C

Type tree for PFB:



Type tree for type of find_p_func:

English: a function that takes
a pointer to char as argument,
and returns a pointer to a function
that takes void as argument and
returns int.



Exercise

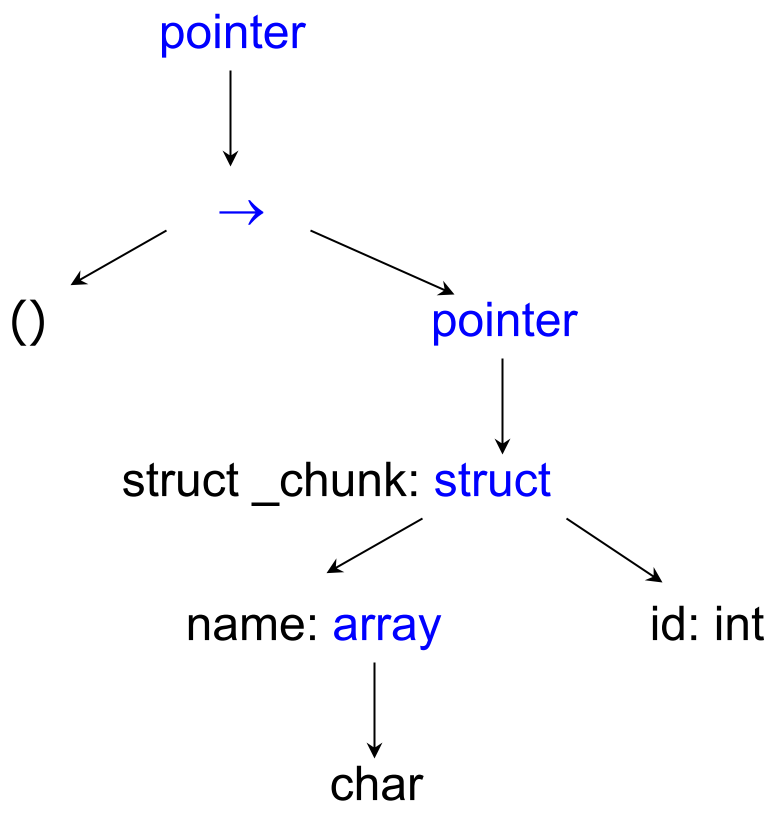
```
struct _chunk {                              // Type struct_chunk: what type?
    char name[10];
    int   id; };
struct obstack {                             // Type struct obstack: what type?
    struct _chunk *chunk;
    struct _chunk *(*chunkfun)();
    void (*freefun) (); };

void chunk_fun(struct obstack *h, void *f) {   // Function chunk_fun: what type?
    h->chunkfun = (struct _chunk *(*)()) f; }
void free_fun(struct obstack *h, void *f) {    // Function free_fun: what type?
    h->freefun = (void (*)()) f; }

int main() {
    struct obstack h;
    chunk_fun(&h,&xmalloc);
    free_fun(&h,&xfree); ... }
```

Declarations in C

Type tree for type of field chunkfun:



Lecture Outline

- Types
- Type systems
  - Type checking
  - Type safety
- Type equivalence
- Types in C

- Primitive types
- Composite types

Primitive Types

- A small collection of built-in types
  - integer, float/real, etc.
- Design issues: e.g., boolean
  - Use integer non-0/0 vs. true/false?
- Implementation issues: representation in the machine
  - Integer
    - Length fixed by standards or implementation (portability issues)
    - Multiple lengths (C: short, int, long)
    - Signs
  - Float/real
    - All issues of integers and more

Composite Types: Record (Struct)

- Collection of heterogeneous fields
- Operations
  - Selection through field names **(s.num, p->next)**
  - Assignment
  - Example: structures in C

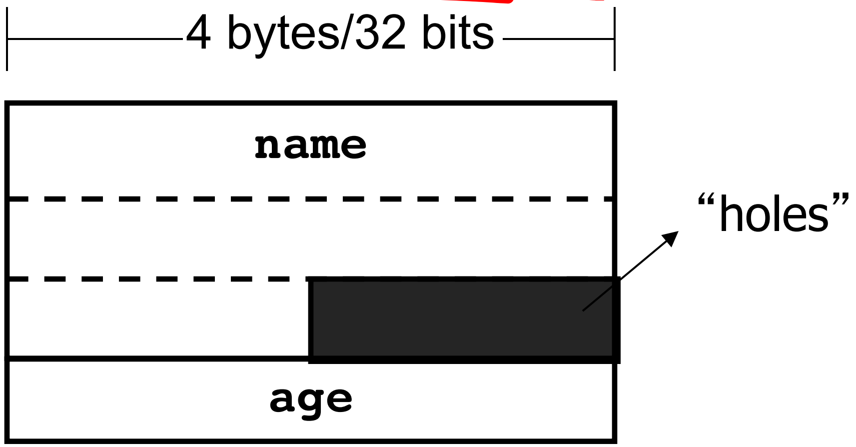
```
typedef struct cell listcell;
struct cell {
        int num;
        listcell *next;
} s,t;
s.num = 0;
s.next = 0;
t = s;
```

Record (Struct)

- Definition of type. What is part of the type?
  - order and type of fields (but not the name)
  - name and type of fields
  - order, name and type of fields
- Implementation issues: memory layout
  - Successive memory locations at offset from first byte. Usually, word-aligned, but sometimes packed

```
typedef struct {
   char name[10];
   int age;
} Person;

Person p;
```



Composite Types: Variant (Union)

- Allow a collection of alternative fields; only one alternative is valid during execution
  - Fortran: equivalence
  - Algol68 and C: unions
  - Pascal: variant records

- Problem: how can we assure type-safety?
  - Pascal and C are not type-safe
  - Algol68 is type-safe! Uses run-time checks
- Usually, alternatives use same storage
  - Mutually exclusive value access

Variants (Unions)

- Example: unions in C

d1:

S → Line

S → Quad

```
union data {
   int k;
   char c;
} d1,d2;
```

  - Operations
    - Selection through field names, Assignment:

```
d1.k = 3; d2 = d1; d2.c = 'b';
```

  - What about type safety?

```
if (n>0) d1.k=5 else d1.c='a';
… d1.k << 2 … // What is the problem?
```

Pascal's Variant Record

```
program main(input,output);
type paytype = (salaried,hourly);
var employee : record
        id : integer;
        dept: integer;
        age : integer;
        case payclass: paytype of
          salaried:
               (monthlyrate : real;
                startdate : integer);
          hourly:
               (rateperhour : real;
                reghours : integer;
                overtime : integer);
        end;
```

Type tag

```
begin
employee.id:=001234;
employee.dept:=12;
employee.age:=38;
employee.payclass:=hourly;
employee.rateperhour:=2.75;
employee.reghours:=40;
employee.overtime:=3;
writeln(employee.rateperhour,
        employee.reghours,
        employee.overtime);
{this should bomb as there is no
    monthlyrate because
    payclass=hourly}
writeln(employee.monthlyrate);
```

Output:

2.750000E+00   40   3

2.750000E+00

Pascal Variant Record

```
type paytype = (salaried,hourly);
var employee : record
        id : integer;
        dept: integer;
        age : integer;
        case payclass: paytype of
          salaried:(
            monthlyrate : real;
            startdate : integer);
          hourly: (
            rateperhour : real;
            reghours : integer;
            overtime : integer);
        end;
```

```
employee.payclass:=salaried;
employee.monthlyrate:=575.0;
employee.startdate:=13085;
{this should bomb as there are no
     rateperhour, etc. because
     payclass=salaried}
writeln(employee.rateperhour,
employee.reghours
employee.overtime);
writeln(employee.monthlyrate);
end.
```

Output:

5.750000E+02 13085 3

5.750000E+02

Composite Types: Array

- Homogeneous, indexed collection of values
- Access to individual elements through subscript

- There are many design choices
  - Subscript syntax
  - Subscript type, element type
  - When to set bounds, compile time or run time?
  - How to initialize?
  - What built-in operations are allowed?

Array

- Definition of type. What is part of the type?
  - bounds/dimension/element type
    - Pascal
  - dimension/element type
    - C, FORTRAN, Algol68

- What is the lifetime of the array?
  - Global lifetime, static shape (in static memory)
  - Local lifetime (in stack memory)
    - Static shape (stored in fixed-length portion of stack frame)
    - Shape bound when control enters a scope
      - (e.g., Ada, Fortran allow definition of array bounds when function is entered; stored in variable-length portion of stack frame)
  - “Global” lifetime, dynamic shape (in heap memory)

Example: Algol68 Arrays

- Array type includes dimension and element type; it does not include bounds

```
[1:12] int month; [1:7] int day;
[0:10,0:10] real matrix;
[-4:10,6:9] real table;
```

row int

(row,row) real

Example - `[1:10] [1:5,1:5] int kinglear;`

- What is the type of `kinglear`? row ((row, row) int)
- What is the type of `kinglear[j]`? (row, row) int
- What is the type of `kinglear[j][1,2]`? int
- `kinglear[1,2,3]` ? X ERROR

Array Addressing

- One dimensional array
  - **`X[low:high]`** each element is E bytes
  - Assuming that elements are stored into consecutive memory locations, starting at address **`addr(X[low])`**, what is the address of **`X[j]`**?

  addr(**`X[low]`**) + (**`j-low`**)*E

  - E.g, let **`X[0:10]`** be an array of reals (4 bytes)
    - **`X[3]?`** is addr(**`X[0]`**) + (**3** - **0**)***4** = addr(**`X`**) + **12**
    - **`X[1]`** is at address addr(**`X[0]`**) + **4**
    - **`X[2]`** is at address addr(**`X[0]`**) + **8**, etc

Array Addressing

- Memory is a sequence of contiguous locations
- Two memory layouts for two-dimensional arrays:
  - Row-major order and column-major order
- Row-major order:
  - **y[0,0], y[0,1], y[0,2], …, y[0,n], y[1,*], y[2,*],…**

- **y[low1:hi1,low2:hi2]** in Algol68, location **y[j,k]** is

addr(**y[low1,low2]**) + (**hi2-low2**+1)*E*(**j-low1**) + (**k-low2**)*E

#locs per row | #rows in front of row **j** | # elements in row **j** in front of element **[j,k]**

Array Addressing

Consider `y[0:2, 0:5] int matrix`.

Assume row-major order and find the address of **`y[1,3]`**.

address of **`y[1,3]`** = addr(**`y[0,0]`**)+(5-0+1)*4*(1-0)+(3-0)*4

6 elements per row

1 row before row 1

3 elements in row 1 before 3

= addr(**`y[0,0]`**)+24+12

= addr(**`y[0,0]`**)+36

- Analogous formula holds for column-major order
- Row-major and column-major layouts generalize to n-dimensional arrays

Composite Types: Pointers

- A variable or field whose value is a reference to some memory location
  - In C: `int *p;`
- Operations
  - Allocation and deallocation of objects on heap
    - `p = malloc(sizeof(int));      free(p);`
  - Assignment of one pointer into another
    - `int *q = p; int *p = &a;`
  - Dereferencing of pointer
    - `*q = 1;`
  - Pointer arithmetic
    - `p + 2`

Pointers: Recursive Types

- A recursive type is a type whose objects may contain objects of the same type
  - Necessary to build linked structures such as linked lists
- Pointers are necessary to define recursive types in languages that use the value model for variables:

```
struct cell {
    int num;
    struct cell *next;
} c
```

Pointers: Recursive Types

- Recursive types are defined naturally in languages that use the reference model for variables:

```
class Cell {
    int num;
    Cell next;

    Cell() { … }
    …
}
```