CMSC 330: Organization of Programming Languages

Type Systems
Implementation Topics
Review: Lambda Calculus

- A lambda calculus expression is defined as

\[ e ::= x \quad \text{variable} \]
\[ | \quad \lambda x.e \quad \text{function} \]
\[ | \quad e \; e \quad \text{function application} \]

- \( \lambda x.e \) is like \((\text{fun } x \to e)\) in OCaml
Review: Beta-Reduction

• Whenever we do a step of beta reduction...
  – \((\lambda x. e_1) \ e_2 \rightarrow e_1[x/e_2]\)
  – ...alpha-convert variables as necessary

• Examples:
  – \((\lambda x. (\lambda x.x)) \ z = (\lambda x. (\lambda y.y)) \ z \rightarrow z \ (\lambda y.y)\)
  – \((\lambda x. \lambda y. x \ y) \ y = (\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z. y \ z\)
The Need for Types

• Consider untyped lambda calculus
  – false = λx.λy.y
  – 0 = λx.λy.y

• Since everything is encoded as a function...
  – We can easily misuse terms
    • false 0 → λy.y
    • if 0 then ...
    • Everything evaluates to some function

• The same thing happens in assembly language
  – Everything is a machine word (a bunch of bits)
  – All operations take machine words to machine words
What is a Type System?

• A type system is some mechanism for distinguishing good programs from bad
  – Good = well typed
  – Bad = ill typed or not typable; has a type error

• Examples
  – 0 + 1  // well typed
  – false 0 // ill-typed; can’t apply a boolean
Static versus Dynamic Typing

• In a static type system, we guarantee at compile time that all program executions will be free of type errors
  – OCaml and C have static type systems

• In a dynamic type system, we wait until runtime, and halt a program (or raise an exception) if we detect a type error
  – Ruby has a dynamic type system
Simply-Typed Lambda Calculus

• $e ::= n \mid x \mid \lambda x : t . e \mid e \; e$
  – We’ve added integers $n$ as primitives
    • Without at least two distinct types (integer and function),
      can’t have any type errors
  – Functions now include the type of their argument

• $t ::= \text{int} \mid t \rightarrow t$
  – $\text{int}$ is the type of integers
  – $t_1 \rightarrow t_2$ is the type of a function that takes arguments
    of type $t_1$ and returns a result of type $t_2$
  – $t_1$ is the domain and $t_2$ is the range
  – Notice this is a recursive definition, so that we can
    give types to higher-order functions
Type Judgments

• We will construct a type system that proves judgments of the form

\[ A \vdash e : t \]

– “In type environment \( A \), expression \( e \) has type \( t \)”

• If for a program \( e \) we can prove that it has some type, then the program type checks
  – Otherwise the program has a type error, and we’ll reject the program as bad
Type Environments

• A type environment is a map from variables names to their types

• $\emptyset$ is the empty type environment

• $A, x:t$ is just like $A$, except $x$ now has type $t$

• When we see a variable in the program, we’ll look up its type in the environment
### Type Rules

\[ e ::= n \mid x \mid \lambda x : t . e \mid e \; e \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>TInt</td>
<td>( A \vdash n : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>TVar</td>
<td>( x : t \in A )</td>
<td>( A \vdash x : t )</td>
</tr>
<tr>
<td>TFun</td>
<td>( A, x : t \vdash e : t' )</td>
<td>( A \vdash \lambda x : t . e : t \rightarrow t' )</td>
</tr>
<tr>
<td>TApp</td>
<td>( A \vdash e : t \rightarrow t' ) ( A \vdash e' : t )</td>
<td>( A \vdash e ; e' : t' )</td>
</tr>
</tbody>
</table>
Example

\[ A = \{ + : \text{int} \to \text{int} \to \text{int} \} \]

\[ B = A, x : \text{int} \]

\[
\begin{align*}
B \vdash + : \text{i} \to \text{i} \to \text{i} & \quad B \vdash x : \text{int} \\
B \vdash + x : \text{int} \to \text{int} & \quad B \vdash 3 : \text{int} \\
B \vdash + x 3 : \text{int} & \quad A \vdash (\lambda x : \text{int}. + x 3) : \text{int} \to \text{int} \\
& \quad A \vdash 4 : \text{int} \\
A \vdash (\lambda x : \text{int}. + x 3) 4 : \text{int}
\end{align*}
\]
The type rules provide a way to reason about programs (i.e., a formal logic)

- The tree of judgments we just saw is a kind of proof in this logic that the program has a valid type

So the *type checking* problem is like solving a jigsaw puzzle

- Can we apply the rules to a program in such a way as to produce a typing proof?
- We can do this automatically
An Algorithm for Type Checking

(Write this in OCaml!)

TypeCheck : type env × expression → type

TypeCheck(A, n) = int
TypeCheck(A, x) = if x in A then A(x) else fail
TypeCheck(A, λx:t.e) =
    let t' = TypeCheck((A, x:t), e) in t → t'
TypeCheck(A, e1 e2) =
    let t1 = TypeCheck(A, e1) in
    let t2 = TypeCheck(A, e2) in
    if dom(t1) = t2 then range(t1) else fail
Type Inference

• We could extend the rules to allow the type checker to deduce the types of every expression in a program even without the annotations
  – This is called *type inference*
  – *Not covered in this class*
Practice

• Reduce the following:
  – \((\lambda x.\lambda y. x \ y \ y) \ (\lambda a. a) \ b\)
  – (or true) (and true false)
  – \((\ast \ 1 \ 2)\) \(\ast \ m \ n = \lambda M.\lambda N.\lambda x.(M \ (N \ x))\)

• Derive and prove the type of:
  – \(A \vdash (\lambda f: \text{int}->\text{int}.\lambda n: \text{int}. f \ n) \ (\lambda x: \text{int}. + 3 \ x) \ 6\)
    \[
    A = \{ + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \} 
    \]
    – \(\lambda x: \text{int}->\text{int}->\text{int}. \lambda y: \text{int}->\text{int}. \lambda z: \text{int}. x \ z \ (y \ z)\)
Review of CMSC 330

• Syntax
  – Regular expressions
  – Finite automata
  – Context-free grammars

• Semantics
  – Operational semantics
  – Lambda calculus

• Implementation
  – Names and scope
  – Evaluation order
  – Concurrency
  – Generics
  – Exceptions
  – Garbage collection
Implementation: Names and Scope
Names and Binding

• Programs use names to refer to things
  – E.g., in \( x = x + 1 \), \( x \) refers to a variable

• A binding is an association between a name and what it refers to to
  – int \( x \);               /* \( x \) is bound to a stack location containing an int */
  – int \( f \) (int) { ... }  /* \( f \) is bound to a function */
  – class \( C \) { ... }      /* \( C \) is bound to a class */
  – let \( x = e_1 \) in \( e_2 \)      /* \( x \) is bound to \( e_1 \) */
Name Restrictions

• Languages often have various restrictions on names to make lexing and parsing easier
  – Names cannot be the same as keywords in the language
  – Sometimes case is restricted
  – Names generally cannot include special characters like ; , : etc
    • Usually names are upper- and lowercase letters, digits, and _ (where the first character can’t be a digit)
Names and Scopes

• Good names are a precious commodity
  – They help document your code
  – They make it easy to remember what names correspond to what entities

• We want to be able to reuse names in different, non-overlapping regions of the code
Names and Scopes (cont’d)

• A *scope* is the region of a program where a binding is active
  – The same name in a different scope can have a different binding

• A name is *in scope* if it's bound to something within the particular scope we’re referring to

• Two names bound to the same object are *aliases*
Example

void w(int i) {
    ...
}
void x(float j) {
    ...
}
void y(float i) {
    ...
}
void z(void) {
    int j;
    char *i;
    ...
}

• i is in scope
  – in the body of w, the body of y, and after the declaration of j in z
  – but all those i’s are different

• j is in scope
  – in the body of x and z
Ordering of Bindings

- Languages make various choices for when declarations of things are in scope
- Generally, all declarations are in scope from the declaration onward
- What about function calls?

```c
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

- What is the result of calling `g()`?
Static Scope

• In static scoping, a name refers to its closest binding, going from inner to outer scope in the program text
  – Languages like C, C++, Java, Ruby, and OCaml are statically scoped

```plaintext
int i;
{
    int j;
    {
        float i;
        j = (int) i;
    }
}
```
Dynamic Scope

- In a language with *dynamic scoping*, a name refers to its closest binding *at runtime*.
Ordering of Bindings

• Back to the example:

```c
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

• What is the result of calling `g()` . . .
  – ... with static scoping?
  – ... with dynamic scoping?
Static vs. Dynamic Scope

**Static scoping**
- Local understanding of function behavior
- Know at compile-time what each name refers to
- A bit trickier to implement

**Dynamic scoping**
- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (just keep a global table of stacks of variable/value bindings)
Namespaces

• Languages have a “top-level” or outermost scope
  – Many things go in this scope; hard to control collisions

• Common solution: add a hierarchy
  – OCaml: Modules
    • List.hd, String.length, etc.
    • open to add names into current scope
  – Java: Packages
    • java.lang.String, java.awt.Point, etc.
    • import to add names into current scope
  – C++: Namespaces
    • namespace f { class g { ... } }, f::g b, etc.
    • using namespace to add names to current scope
Free and Bound Variables

- The *bound variables* of a scope are those names that are declared in it.
- If a variable is not bound in a scope, it is *free*
  - The bindings of variables which are free in a scope are "inherited" from declarations of those variables in outer scopes in static scoping.

```c
{ /* 1 */
  int j;
  { /* 2 */
    float i;
    j = (int) i;
  }
}
```

- `j` is bound in scope 1.
- `j` is free in scope 2.
- `i` is bound in scope 2.
Implementation: Evaluation Order
Call-by-Value (CBV)

• In *call-by-value* (*eager evaluation*), arguments to functions are fully evaluated before the function is invoked
  – This is the standard evaluation order that we're used to from C, C++, and Java
  – Also in OCaml, in `let x = e1 in e2`, the expression `e1` is fully evaluated before `e2` is evaluated
Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature
  - What does this program print?

```c
void f(int x) {
    x = 3;
}

int main() {
    int x = 0;
    f(x);
    printf("%d\n", x);
}
```

- Prints 0
Call-by-Value in Imperative Languages, con't.

- Actual parameter is copied to stack location of formal parameter

```c
void f(int x) {
    x = 3;
}
int main() {
    int x = 0;
    f(x);
    printf("%d\n", x);
}
```

- Modification of formal parameter not reflected in actual parameter!
Call-by-Reference (CBR)

- Alternative idea: Implicitly pass a *pointer* or *reference* to the actual parameter
  - If the function writes to it the actual parameter is modified

```c
void f(int x) {
    x = 3;
}

int main() {
    int x = 0;
    f(x);
    printf("%d\n", x);
}
```
Call-by-Reference (cont’d)

• Advantages
  – The entire argument doesn't have to be copied to the called function
    • It's more efficient if you’re passing a large (multi-word) argument
  – Allows easy multiple return values

• Disadvantages
  – Can you pass a non-variable (e.g., constant, function result) by reference?
  – It may be hard to tell if a function modifies an argument
  – What if you have aliasing?
Aliasing

• We say that two names are aliased if they refer to the same object in memory
  – C examples (this is what makes optimizing C hard)

```c
int x;
int *p, *q; /*Note that C uses pointers to simulate call by reference */
p = &x; /* *p and x are aliased */
q = p; /* *q, *p, and x are aliased */
```

```c
struct list { int x; struct list *next; }  
struct list *p, *q;
...
q = p; /* *q and *p are aliased */
/* so are p->x and q->x */
/* and p->next->x and q->next->x... */
```
Call-by-Name (CBN)

• *Call-by-name* (lazy evaluation)
  – *Theory simple*: In a function:
    
    ```
    Let add x y = x+y
    add (a*b) (c*d)
    ```

    Then each use of *x* and *y* in the function definition is just a literal substitution of the actual arguments, *(a*b)* and *(c*d)*, respectively
  
  – *Implementation difficult*: Highly complex, inefficient, and provides little improvement over other mechanisms, as later slides demonstrate
Call-by-Name (cont’d)

- In *call-by-name*, arguments to functions are evaluated at the last possible moment, just before they're needed.

\[
\text{let add } x \ y = x + y \\
\text{let } z = \text{add } (\text{add} \ 3 \ 1) \ (\text{add} \ 4 \ 1)
\]

OCaml; cbv; arguments evaluated here

\[
\text{add } x \ y = x + y \\
\text{z = add } (\text{add} \ 3 \ 1) \ (\text{add} \ 4 \ 1)
\]

Haskell; cbn; arguments evaluated here
Call-by-Name (cont’d)

• What would be an example where this difference matters?

```ocaml
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; eager; infinite recursion at call

```ocaml
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; lazy; never evaluated because parameter is never used
Three-Way Comparison

• Consider the following program under the three calling conventions
  – For each, determine i's value and which a[i] (if any) is modified

```c
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
p(a[i], i);
    printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```
Example: Call-by-Value

```c
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d
", i, a[0], a[1], a[2]);
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: Call-by-Reference

```c
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

```
1 0 1 2
2 10
2 10
```

Example: Call-by-Name

```c
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
    a[i] = 5*i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d
", i, a[0], a[1], a[2]);
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

The expression `a[i]` isn't evaluated until needed, in this case after `i` has changed.
Evaluation Order in Lambda Calculus

\((\lambda x. x) \ ((\lambda y. y \ y) \ z)\)
Evaluation Order in Lambda Calculus

\[(\lambda x. x) ((\lambda y. y \ y) \ z)\]
\[\quad (\lambda x. x) (z \ z)\]
\[\quad z \ z\]

Eager

\[(\lambda x. x) ((\lambda y. y \ y) \ z)\]
\[\quad (\lambda y. y \ y) \ z\]
\[\quad z \ z \ z\]

Lazy
CBV versus CBN

• CBN is flexible- strictly more programs terminate
  – E.g., where we might have an infinite loop with cbv, we might avoid it with cbn by waiting to evaluate

• Order of evaluation is really hard to see in CBN
  – Call-by-name doesn't mix well with side effects (assignments, print statements, etc.)

• Call-by-name is more expensive since:
  – Functions have to be passed around
  – If you use a parameter twice in a function body, its thunk (the unevaluated argument) will be called twice
  • Haskell actually uses call-by-need (each formal parameter is evaluated only once, where it's first used in a function)
Review

• Evaluation strategies
  – Names and bindings
    • Free vs. bound
  – Scope
    • Static vs. dynamic
  – Reduction order
    • Eager evaluation
    • Lazy evaluation
  – Parameter passing
    • Call-by-value
    • Call-by-reference
    • Call-by-name
    • and others...

(calling convention)
Implementation: Function Calls
How Function Calls Really Work

• Function calls are so important they usually have direct instruction support on the hardware

• We won’t go into the details of assembly language programming
  – See CMSC 212, 311, 412, or 430

• But we will discuss just enough to know how functions are called
Machine Model (x86)

• The CPU has a fixed number of registers
  – Think of these as memory that’s really fast to access
  – For a 32-bit machine, each can hold a 32-bit word

• Important x86 registers
  – eax   generic register for computing values
  – esp   pointer to the top of the stack
  – ebp   pointer to start of current stack frame
  – eip   the program counter (points to next instruction in text segment to execute)
The x86 Stack Frame/Activation Record

- The stack just after \( f \) transfers control to \( g \)

Based on Fig 6-1 in Intel ia-32 manual
x86 Calling Convention

• To call a function
  – Push parameters for function onto stack
  – Invoke CALL instruction to
    • Push current value of eip onto stack
      – I.e., save the program counter
    • Start executing code for called function
  – Callee pushes ebp onto stack to save it

• When a function returns
  – Put return value in eax
  – Invoke LEAVE to pop stack frame
    • Set esp to ebp
    • Restore ebp that was saved on stack and pop it off the stack
  – Invoke RET instruction to load return address into eip
    • I.e., start executing code where we left off at call
Example

```c
int f(int a, int b) {
    return a + b;
}

int main(void) {
    int x;
    x = f(3, 4);
}
```

gcc -S a.c

```assembly
f:
    pushl  %ebp
    movl  %esp, %ebp
    movl  12(%ebp), %eax
    addl  8(%ebp), %eax
    leave
    ret

main:
    ...
    subl  $8, %esp
    pushl $4
    pushl $3
    call  f
l:
    addl  $16, %esp
    movl  %eax, -4(%ebp)
    leave
    ret
```
Lots More Details

• There’s a whole lot more to say about calling functions
  – Local variables are allocated on stack by the callee as needed
    • This is usually the first thing a called function does
  – Saving registers
    • If the callee is going to use eax itself, you’d better save it to the stack before you call
  – Passing parameters in registers
    • More efficient than pushing/popping from the stack
  – Etc...

• See other courses for more details
Tail Calls

• A *tail call* is a function call that is the last thing a function does before it returns.

```ocaml
let add x y = x + y
let f z = add z z (* tail call *)

let rec length = function
    [] -> 0
| (_::t) -> 1 + (length t) (* not a tail call *)

let rec length a = function
    [] -> a
| (_::t) -> length (a + 1) t (* tail call *)
```
Tail Recursion

• Recall that in OCaml, all looping is via recursion
  – Seems very inefficient
  – Needs one stack frame for recursive call

• A function is *tail recursive* if it is recursive and the recursive call is a tail call
Tail Recursion (cont’d)

However, if the program is tail recursive...
  – Can instead reuse stack frame for each recursive call

```c
let rec length l = match l with
  | [] -> 0
  | (_::t) -> 1 + (length t)
length [1;2]
```

eax: 2
Tail Recursion (cont’d)

- The same stack frame is reused for the next call, since we’d just pop it off and return anyway

```ocaml
let rec length a l = match l with
  | [] -> a
  | (_::t) -> (length (a + 1) t)
length 0 [1; 2]
```

 eax: 2
Tail Recursion (cont'd)

- Corollary: any tail-recursive function can be rewritten using an iterative loop.

```c
int fact2 (int n, int a) {
    if (n<=1)
        return a;
    else
        return fact(n-1, a*n);
}

int fact (int n) {
    int v = 1;
    while (int n >= 1)
        v *= n--;
    return v;
}
```

```c
int fact (int n) {
    int v = 1;
    while (int n >= 1)
        v *= n--;
    return v;
}
```