• **HW4**: OAT v. 1.0
  – Parsing & basic code generation
Why Inference Rules?

- Allow a compact, precise way of specifying language properties
  - About 20 pages for full Java, versus
  - Hundreds of pages of prose Java Language Specification

- Correspond closely to recursive AST traversal for implementing them

- Type checking/inference tries to prove a different judgment $C \vdash e : t$
  - By searching backward through the rules

- Compiling is “interpreting” the type checking rules $[C \vdash e : t]$;
  - Compilation follows the type checking judgment

- Strong mathematical foundations, e.g., “Curry-Howard correspondence”
  - Programming Language ~ Logic
  - Program ~ Proof
  - Type ~ Proposition
OPTIMIZING CONTROL
Standard Evaluation

• Consider compiling the following program fragment

```c
if (x & !y | !w)
    z = 3;
else
    z = 4;
return z;
```

```assembly
%tmp1 = icmp Eq [y], 0 ; !y
%tmp2 = and [x] %tmp1
%tmp3 = icmp Eq [w], 0
%tmp4 = or %tmp2, %tmp3
%tmp5 = icmp Eq %tmp4, 0
br %tmp5, label %else, label %then

then:
    store [z], 3
    br %merge

else:
    store [z], 4
    br %merge

merge:
    %tmp6 = load [z]
    ret %tmp6
```
Observation

• Usually, we want the translation $[e]$ to produce a value
  – $[C \vdash e : t] = (\text{ty}, \text{operand}, \text{stream})$
  – e.g. $[C \vdash e_1 + e_2 : \text{int}] = (\text{i64}, \%\text{tmp}, [\%\text{tmp} = \text{add } [e_1] [e_2]])$

• But, when the compiled expression appears in a test
  – The program jumps to one label or another after the comparison
  – Otherwise, it never uses the value

• In many cases, we can avoid “materializing” the value (i.e. storing it in a temporary) and thus produce better code
  – This idea also lets us implement different functionality too: e.g. short-circuiting Boolean expressions
Idea: Use a different translation for tests

- Usual Expression translation: $[C \vdash e : t] = (ty, \text{operand}, \text{stream})$

- Conditional branch translation of Booleans, without materializing value
  $[C \vdash e : \text{bool}@] \text{ ltrue lfalse} = \text{stream}$

Notes
- Two extra arguments
  - “true” branch label
  - “false” branch label
- Doesn’t “return a value”

- Aside: this is a form of continuation-passing translation

where
\[
\begin{align*}
[C, \text{rt} \vdash s_1 \Rightarrow C'] &= [C'], \text{insns}_1 \\
[C, \text{rt} \vdash s_2 \Rightarrow C'] &= [C'], \text{insns}_2 \\
[C \vdash e : \text{bool}@] \text{ then else} &= \text{insns}_3
\end{align*}
\]
Short Circuit Compilation: Expressions

- $\llbracket C \vdash e : \text{bool}@ \rrbracket \; \text{ltrue lfalse} = \text{insns}$

\[
\begin{align*}
\llbracket C \vdash \text{false} : \text{bool}@ \rrbracket \; \text{ltrue lfalse} &= \; [\text{br} \; \%lfalse] & \text{FALSE} \\
\llbracket C \vdash \text{true} : \text{bool}@ \rrbracket \; \text{ltrue lfalse} &= \; [\text{br} \; \%ltrue] & \text{TRUE}
\end{align*}
\]

- $\llbracket C \vdash e : \text{bool}@ \rrbracket \; \text{lfalse ltrue} = \text{insns}$

- $\llbracket C \vdash !e : \text{bool}@ \rrbracket \; \text{ltrue lfalse} = \text{insns}$
Short Circuit Evaluation

Idea: build the logic into the translation

\[
\begin{align*}
\llbracket C \vdash e_1 : \text{bool}@ \rrbracket \ ltrue \ \text{right} &= \ insns_1 \\
\llbracket C \vdash e_2 : \text{bool}@ \rrbracket \ ltrue \ lfalse &= \ insns_2
\end{align*}
\]

where \text{right} is a fresh label
Consider compiling the following program fragment:

```c
if (x & !y | !w) 
    z = 3;
else 
    z = 4;
return z;
```

```assembly
%tmp1 = icmp Eq [x], 0
br %tmp1, label %right2, label %right1

right1:
    %tmp2 = icmp Eq [y], 0
    br %tmp2, label %then, label %right2

right2:
    %tmp3 = icmp Eq [w], 0
    br %tmp3, label %then, label %else

then:
    store [z], 3
    br %merge

else:
    store [z], 4
    br %merge

merge:
    %tmp5 = load [z]
    ret %tmp5
```
Compiling lambda calculus to straight-line code.
Representing evaluation environments at runtime.

CLOSURE CONVERSION
“Functional” languages

• Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
  – Functions can be passed as arguments (e.g. map or fold)
  – Functions can be returned as values (e.g. compose)
  – Function nest:
    Inner function can refer to variables bound in the outer function

let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec

• How do we implement such functions?
  – In an interpreter? In a compiled language?
Compiling First-class Functions

• To implement first-class functions on a processor, there are 2 problems
  – Must implement substitution of free variables
  – Must separate “code” from “data”

• Reify the substitution
  – Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  – The environment-based interpreter is one step in this direction

• Closure conversion
  – Eliminates free variables by packaging up the needed environment in the data structure

• Hoisting
  – Separates code from data, pulling closed code to the top level
Example of Closure Creation

• Recall the “add” function
  \[
  \text{let add = fun } x \rightarrow \text{ fun } y \rightarrow x + y
  \]

• Consider the inner function: \( \text{fun } y \rightarrow x + y \)

• When run the function application: \( \text{add 4} \)
  the program builds a closure and returns it
  – The closure is a pair of the environment and a code pointer
    - The function code is (essentially)
      \[
      \text{fun (env, y) } \rightarrow \text{ let } x = \text{nth env 0 in } x + y
      \]
Representing Closures

• The simple closure conversion doesn’t generate very efficient code
  – It stores all the values for variables in the environment, even if they aren’t needed by the function body
  – It copies the environment values each time a nested closure is created
  – It uses a linked-list data structure for tuples

• There are many options
  – Store only the values for free variables in the body of the closure
  – Share subcomponents of the environment to avoid copying
  – Use vectors or arrays rather than linked structures
Array-based Closures with N-ary Functions

\[(\text{fun } (x \ y \ z) \rightarrow (\text{fun } (n \ m) \rightarrow (\text{fun } p \rightarrow (\text{fun } q \rightarrow n + z) \ x))\]

Note how free variables are “addressed” relative to the closure due to shared env.
Scope, Types, and Context

STATIC ANALYSIS
Adding Integers to Lambda Calculus

\[\text{exp} ::= \]
\[\quad | \ldots\]
\[\quad | \text{n} \quad \text{constant integers}\]
\[\quad | \text{exp}_1 + \text{exp}_2 \quad \text{binary arithmetic operation}\]

\[\text{val} ::= \]
\[\quad | \text{fun} \ x \rightarrow \text{exp} \quad \text{functions are values}\]
\[\quad | \text{n} \quad \text{integers are values}\]

\[\text{n}\{v/x\} = \text{n} \quad \text{constants have no free vars}\]

\[\text{(e}_1 + \text{e}_2)\{v/x\} = (\text{e}_1\{v/x\} + \text{e}_2\{v/x\}) \quad \text{substitute everywhere}\]

\[\text{NOTE: there are no rules for the case where } \text{exp}_1 \text{ or } \text{exp}_2 \text{ evaluate to functions! The semantics is undefined in those cases.}\]

\[\text{exp}_1 \downarrow \text{n}_1 \quad \text{exp}_2 \downarrow \text{n}_2\]

\[\text{exp}_1 + \text{exp}_2 \downarrow (\text{n}_1 \ [+] \text{n}_2)\]

object-level ‘+’  meta-level ‘+’
Variable Scoping

• Problem: How to determine whether a declared variable is in scope?

• Issues
  – Q1: Which variables are available at a given program location?
  – Q2: Can the same identifier be reused (i.e., shadowing), or it is an error?

• Example: code below is syntactically correct, but not well-formed
  – y and q are used without being defined anywhere

```c
int fact(int x) {
    var acc = 1;
    while (x > 0) {
        acc = acc * y;
        x = q - 1;
    }
    return acc;
}
```

Q: Can we solve this problem by changing the parser to rule out such programs?
The interpreter from the Eval3 module (fun.ml, Lec 13)

```ml
let rec eval env e =
  match e with
  | ... |
  | Add (e1, e2) ->
    (match (eval env e1, eval env e2) with
     | (IntV i1, IntV i2) -> IntV (i1 + i2)
     | _ -> failwith "tried to add non-integers")
  | ... |
```

- The interpreter might fail at runtime
  - Not all operations are defined for all values (e.g. 3/0, 3 + true, ...)
- A compiler can’t generate sensible code for this case
  - A naïve implementation might “add” an integer and a function pointer
See tc.ml (lec13.zip)
Notes about this Typechecker

• The interpreter only evaluates the body of a function when it's applied

• Typechecker always check function’s body (even if it's never applied)
  – Assume the input has some type (say \( t_1 \))
  – Reflect this information in the type of the function (\( t_1 \rightarrow t_2 \))

• Dually, at a call site (\( e_1 \ e_2 \)), we don't know what closure we’ll get, but
  – Can calculate \( e_1 \)'s type
  – Check \( e_2 \) is an argument of the right type
  – Determine what type \( e_1 \) will return

(Q1) Why is this an approximation?

(Q2) What if well_typed always returns false?
Contexts and Inference Rules

• Need to keep track of contextual information
  – What variables are in scope?
  – What are their types?
  – What information do we have about each syntactic construct?

• What relationships are there among the syntactic objects?
  – e.g. is one type a subtype of another?

• How do we describe this information?
  – In the compiler, there's a mapping from variables to information we know about them – the "context"
  – The compiler has a collection of (mutually recursive) functions that follow the structure of the syntax
Type Judgments

• In the judgment: \( E \vdash e : t \)
  - \( E \) is a *typing environment* or a *type context*
  - \( E \) maps variables to types and is simply a set of bindings of the form:
    \( x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n \)

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

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

For the language in “tc.ml”, we have five inference rules

- **INT**
  - \[E \vdash i : \text{int}\]

- **VAR**
  - \[x : T \in E \quad \rightarrow \quad E \vdash x : T\]

- **ADD**
  - \[E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int} \quad \rightarrow \quad E \vdash e_1 + e_2 : \text{int}\]

- **FUN**
  - \[E, x : T \vdash e : S \quad \rightarrow \quad E \vdash \text{fun} (x:T) -> e : T -> S\]

- **APP**
  - \[E \vdash e_1 : T -> S \quad E \vdash e_2 : T \quad \rightarrow \quad E \vdash e_1 e_2 : S\]

Note how these rules correspond to the code
Type Checking Derivations

- **Derivation** or *proof tree*
  - Nodes: judgments
  - Edges: connect premises to a conclusion (according to inference rules)
- Leaves of the tree are *axioms* (i.e., rules with no premises)
  - Example: the INT rule is an axiom
- **Goal of the type checker**: verify that such a tree exists

- Ex: Find a tree for the following code using the given inference rules

\[ \vdash \text{(fun } (x:\text{int}) \to x + 3) \text{ 5 : int} \]
Example Derivation Tree

\[
x : \text{int} \in x : \text{int}
\]

```
<table>
<thead>
<tr>
<th>VAR</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>x : int ⊢ x : int</td>
<td>x : int ⊢ 3 : int</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ADD</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x : int ⊢ x + 3 : int</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>FUN</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ (fun (x:int) -&gt; x + 3) : int -&gt; int</td>
<td>⊢ 5 : int</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>APP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ (fun (x:int) -&gt; x + 3) 5 : int</td>
<td></td>
</tr>
</tbody>
</table>
```

Note
- The OCaml function `typecheck` verifies the existence of this tree
- Recursive calls for running `typecheck` follow the same shape as this tree
- \( x : \text{int} \in E \) is implemented by the function `lookup`
Type Safety

"Well typed programs do not go wrong."
– Robin Milner, 1978

**Theorem:** (simply typed lambda calculus with integers)

\[
\text{If } \vdash e : t, \text{ then there exists a value } v \text{ such that } e \Downarrow v
\]

- **Note:** This is a very strong property
  - Well-typed programs never executes undefined code like 3 + (fun x -> 2)
  - Simply-typed lambda calculus terminates (i.e., not Turing complete)
**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 behavior
  - 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
Arrays

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

```
NEW
E ⊢ e_1 : int       E ⊢ e_2 : T
_________________
E ⊢ new T[e_1](e_2) : T[]
```

```
INDEX
E ⊢ e_1 : T[]       E ⊢ e_2 : int
_________________
E ⊢ e_1[e_2] : T
```

```
UPDATE
E ⊢ e_1 : T[]       E ⊢ e_2 : int       E ⊢ e_3 : T
_________________
E ⊢ e_1[e_2] = e_3 ok
```

$e_1$: size of newly alloc. array
$e_2$: initializes the array

Note: These rules don't ensure array indices are within bounds, which should be checked dynamically.
Tuples

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

**TUPLE**

\[
E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n
\]

\[
E \vdash (e_1, \ldots, e_n) : T_1 \ast \ldots \ast T_n
\]

**PROJ**

\[
E \vdash e : T_1 \ast \ldots \ast T_n \quad 1 \leq i \leq n
\]

\[
E \vdash \#i e : T_i
\]
ML-style references (note that ML uses only expressions)
• First, add a new type constructor: \( T \text{ ref} \)

\[
\begin{align*}
E \vdash e : T \\
E \vdash \text{ref } e : T \text{ ref}
\end{align*}
\]

\[
\begin{align*}
E \vdash e : T \text{ ref} \\
E \vdash !e : T
\end{align*}
\]

\[
\begin{align*}
E \vdash e_1 : T \text{ ref} & \quad E \vdash e_2 : T \\
\hline
E \vdash e_1 := e_2 : \text{unit}
\end{align*}
\]

Note the similarity with the rules for arrays
Beyond describing “structure”… describing “properties”
Types as sets
Subsumption

TYPES, MORE GENERALLY
What are types, anyway?

• A type is just a predicate on the set of values in a system
  – E.g., the type “int” can be thought of as a boolean function that returns “true” on integers and “false” otherwise
  – Equivalently, we can think of a type as just a subset of all values

• For efficiency and tractability, the predicates are usually very simple
  – Types are an abstraction mechanism

• We can easily add new types that distinguish different subsets of values

type tp =
  | IntT (* type of integers *)
  | PosT | NegT | ZeroT (* refinements of ints *)
  | BoolT (* type of booleans *)
  | TrueT | FalseT (* subsets of booleans *)
  | AnyT (* any value *)
Modifying the typing rules

- We need to refine the typing rules too
- Some easy cases
  - Just split up the integers into their more refined cases

\[
\begin{align*}
P-\text{INT} & & N-\text{INT} & & \text{ZERO} \\
\text{i} > 0 & & \text{i} < 0 & & \\
\hline
E \vdash \text{i} : \text{Pos} & & E \vdash \text{i} : \text{Neg} & & E \vdash 0 : \text{Zero}
\end{align*}
\]

- Same for booleans

\[
\begin{align*}
\text{TRUE} & & \text{FALSE} \\
\hline
E \vdash \text{true} : \text{True} & & E \vdash \text{false} : \text{False}
\end{align*}
\]
What about “if”?

• Two cases are easy

\[
\begin{align*}
\text{IF-T} & \quad E \vdash e_1 : \text{True} \quad E \vdash e_2 : T \\
\text{IF-F} & \quad E \vdash e_1 : \text{False} \quad E \vdash e_3 : T
\end{align*}
\]

\[E \vdash \text{if } (e_1) \text{ e}_2 \text{ else e}_3 : T\]

• What if we don’t know statically which branch will be taken?
• Consider the typechecking problem

\[x : \text{bool} \vdash \text{if } (x) \ 3 \ \text{else } -1 : ?\]

• The true branch has type Pos, while the false branch has type Neg
  – What should be the result type of the whole if?
Subtyping and Upper Bounds

• If we view types as sets of values, there is a natural inclusion relation
  \( \text{Pos} \subseteq \text{Int} \)

• This subset relation gives rise to a subtype relation: \( \text{Pos} <: \text{Int} \)

• Such inclusions give rise to a subtyping hierarchy

\[
\begin{array}{c}
\text{Any} \\
\text{Int} & \text{Bool} \\
\text{Neg} & \text{Zero} & \text{Pos} & \text{True} & \text{False}
\end{array}
\]

• For types \( T_1, T_2 \), define their least upper bound (LUB) wrt the hierarchy
  – Example: \( \text{LUB}(\text{True}, \text{False}) = \text{Bool} \), \( \text{LUB}(\text{Int}, \text{Bool}) = \text{Any} \)
  – Note: may want to add types for “NonZero”, “NonNegative”, “NonPositive”,
    so that set union on values corresponds to taking LUBs on types
For statically unknown conditionals, we want the return value to be the LUB of the types of the branches.

\[
E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2
\]

\[
E \vdash \text{if } (e_1) \text{ e}_2 \text{ else e}_3 : \text{LUB}(T_1, T_2)
\]

Note LUB($T_1$, $T_2$) is the most precise type (wrt the hierarchy) that is able to describe any value that has either type $T_1$ or type $T_2$.

In math notation, LUB($T_1$, $T_2$) is sometimes written $T_1 \lor T_2$.

LUB is also called the join operation.
Subtyping Hierarchy

• A subtyping hierarchy

```
Subtyping Hierarchy

• A subtyping hierarchy

<: Int Pos True
<: Neg Zero False
<: Any
<: Bool

The subtyping relation is a partial order

– Reflexive: T <: T for any type T
– Transitive: T_1 <: T_2 and T_2 <: T_3 then T_1 <: T_3
– Antisymmetric: If T_1 <: T_2 and T_2 <: T_1 then T_1 = T_2
```
Soundness of Subtyping Relations

• We don’t have to treat every subset of the integers as a type
  – e.g., we left out the type NonNeg

• A subtyping relation $T_1 <: T_2$ is \textit{sound} if it approximates the underlying semantic subset relation

• Formally: write $⟦T⟧$ for the subset of (closed) values of type $T$
  – i.e. $⟦T⟧ = \{v \mid \vdash v : T\}$
  – e.g. $⟦\text{Zero}⟧ = \{0\}$, $⟦\text{Pos}⟧ = \{1, 2, 3, \ldots\}$

• If $T_1 <: T_2$ implies $⟦T_1⟧ \subseteq ⟦T_2⟧$, then $T_1 <: T_2$ is sound
  – Ex 1: $\text{Pos} <: \text{Int}$ is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  – Ex 2: $\text{Int} <: \text{Pos}$ is not sound, since it is \textit{not} the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$
Soundness of LUBs

• Whenever we have a sound subtyping relation, it follows that
  \[ \llbracket \text{LUB}(T_1, T_2) \rrbracket \supseteq \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket \]
  
  – Note that the LUB is an over approximation of the “semantic union”
  – Example: \[ \llbracket \text{LUB}(\text{Zero}, \text{Pos}) \rrbracket = \llbracket \text{Int} \rrbracket = \{…, -3, -2, -1, 0, 1, 2, 3, …\} \supseteq \{0, 1, 2, 3, …\} = \llbracket \text{Zero} \rrbracket \cup \llbracket \text{Pos} \rrbracket \]

• Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule)

• It just so happens that LUBs on types <: Int correspond to +

\[
\begin{align*}
\text{E} &\vdash e_1 : T_1 \quad \text{E} \vdash e_2 : T_2 \\
T_1 &\text{ <: Int} \quad T_2 \text{ <: Int} \\
\hline
\text{E} &\vdash e_1 + e_2 : T_1 \lor T_2
\end{align*}
\]
Subsumption Rule

• When we add subtyping judgments of the form $T <: S$ we can uniformly integrate it into the type system generically

$$E \vdash e : T \quad T <: S$$

$$\overline{\quad}$$

$$E \vdash e : S$$

• Subsumption allows any value of type $T$ to be treated as an $S$ whenever $T <: S$

• Adding this rule makes the search for typing derivations more difficult – this rule can be applied anywhere, since $T <: T$
  – But careful engineering of the typing system can incorporate the subsumption rule into a deterministic algorithm
  – See for example the OAT type system
Downcasting

• What happens if we have an Int, but need something of type Pos?
  – At compile time, we don’t know whether the Int is greater than zero
  – At run time, we do

• Add a “checked downcast”

\[
E \vdash e_1 : \text{Int} \quad E, x : \text{Pos} \vdash e_2 : T_2 \quad E \vdash e_3 : T_3
\]

\[
E \vdash \text{ifPos} (x = e_1) \ e_2 \ \text{else} \ e_3 : T_2 \lor T_3
\]

• At runtime, \text{ifPos} checks whether \(e_1\) is > 0. If so, branches to \(e_2\) and otherwise branches to \(e_3\)
• Inside the expression \(e_2\), \(x\) is the name for \(e_1\)’s value, which is known to be strictly positive because of the dynamic check
• Note such rules force the programmer to add the appropriate checks
  – We could give integer division the type: \(\text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int}\)