Announcements

• HW4: Due last night

• HW5: Oat v2. Typechecking and Safety
  – due: Friday, April 17th
  – Available soon
  – Start Early!

• HW6: now (tentatively) due: Wednesday, April 29th
Inference Rules

- We can read a judgment $G;L \vdash e : t$ as “the expression $e$ is well typed and has type $t$”
- For any environment $G$, expression $e$, and statements $s_1$, $s_2$.

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

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

- More succinctly: we summarize these constraints as an inference rule:

Premises

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$G;L \vdash e : \text{bool}$</td>
<td>$G;L;rt \vdash s_1$</td>
<td>$G;L;rt \vdash s_2$</td>
</tr>
</tbody>
</table>

Conclusion

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$G;L;rt \vdash \text{if (e)} s_1 \text{ else } s_2$</td>
<td></td>
</tr>
</tbody>
</table>

- This rule can be used for any substitution of the syntactic metavariables $G$, $e$, $s_1$ and $s_2$. 
Checking Derivations

• A derivation or proof tree has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.

• 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.

• Example1: Find a tree for the following program using the inference rules in oat-v1-defn.pdf:

```plaintext
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

Example2: There is no tree for this ill-scoped program:

```plaintext
var x2 = x1 + x1;
return(x2);
```
Example Derivation

\[
\begin{align*}
&D_1 \quad D_2 \quad D_3 \quad D_4 \\
G_0; \cdot ; \text{int} \vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; \Rightarrow \cdot , x_1 : \text{int}, x_2 : \text{int} \\
\vdash \text{var } x_1 = 0; \text{var } x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1;
\end{align*}
\]
Example Derivation

\[
\begin{array}{c}
\frac{\Gamma \vdash 0 : \text{int}}{\Gamma} \quad \text{[INT]} \\
\frac{\Gamma \vdash 0 : \text{int}}{\Gamma} \quad \text{[CONST]} \\
\frac{\Gamma \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1 : \text{int}}{\cdot ; \text{int} \vdash \text{var } x_1 = 0; \Rightarrow \cdot, x_1 : \text{int}} \quad \text{[SDECL]} \\
\end{array}
\]

\[
\mathcal{D}_1 = \frac{\cdot ; \text{int} \vdash \text{var } x_1 = 0; \Rightarrow \cdot, x_1 : \text{int}}{\Gamma ; \cdot \vdash \cdot, x_1 : \text{int}} \quad \text{[SDECL]}
\]

\[
\begin{array}{c}
\frac{\vdash + : (\text{int, int}) \rightarrow \text{int}}{\cdot \vdash + : (\text{int, int}) \rightarrow \text{int}} \quad \text{[ADD]} \\
\frac{\cdot ; x_1 : \text{int} \vdash x_1 : \text{int}}{\cdot ; x_1 : \text{int} \vdash x_1 : \text{int}} \quad \text{[VAR]} \\
\frac{\cdot ; x_1 : \text{int} \vdash x_1 : \text{int}}{\cdot ; x_1 : \text{int} \vdash x_1 : \text{int}} \quad \text{[VAR]} \\
\frac{\Gamma ; \cdot, x_1 : \text{int} \vdash x_1 + x_1 : \text{int}}{\cdot , x_1 : \text{int} \vdash x_1 + x_1 : \text{int}} \quad \text{[BOP]} \\
\frac{\cdot , x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}}{\cdot , x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}} \quad \text{[SDECL]} \\
\end{array}
\]

\[
\mathcal{D}_2 = \frac{\cdot , x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}}{\cdot ; x_1 : \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}} \quad \text{[SDECL]}
\]
Example Derivation

\[ D_3 \]

\[
\begin{array}{l}
    x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int} ; \\
    \vdash -: (\text{int, int}) \rightarrow \text{int} [\text{ADD}] \\
    \frac{\vdash x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_1: \text{int} [\text{VAR}]} \\
    \frac{\vdash x_2: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_2: \text{int} [\text{VAR}]} \\
    \frac{\vdash x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_1 - x_2: \text{int} [\text{BOP}]} \\
    \frac{\vdash -: (\text{int, int}) \rightarrow \text{int} [\text{ADD}]}{\vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} [\text{ASSN}]} \\
\end{array}
\]

\[ D_4 \]

\[
\begin{array}{l}
    \frac{\vdash x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash x_1: \text{int} [\text{VAR}]} \\
    \frac{\vdash x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}, x_2: \text{int} \vdash \text{return } x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} [\text{RET}]} \\
\end{array}
\]
Why Inference Rules?

• They are a 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.

• Compiling in a context is nothing more an “interpretation” of the inference rules that specify typechecking*:
  \[ [C \vdash e : t] \]
  – Compilation follows the typechecking judgment.

• Strong mathematical foundations
  – The “Curry-Howard correspondence”: Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  – See CIS 500 next Fall if you’re interested in type systems!

*Here (and later) we’ll write context C for \(G; L\), the combination of the global and local contexts.
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?

\[ \llbracket C \vdash e : t \rrbracket = ? \]

• \[ \llbracket t \rrbracket \] is a target type
• \[ \llbracket e \rrbracket \] translates to a (potentially empty) sequence of instructions, that, when run, computes the result into some operand

• INVARIANT: if \[ \llbracket C \vdash e : t \rrbracket = ty, \text{operand}, \text{stream} \] then the type (at the target level) of the operand is \( ty = \llbracket t \rrbracket \)
Example

- $C \vdash 341 + 5 : \text{int}$  what is $\llbracket C \vdash 341 + 5 : \text{int} \rrbracket$ ?

\[
\begin{align*}
\llbracket \vdash 341 : \text{int} \rrbracket &= (\text{i64}, \text{Const 341}, []) \\
\llbracket \vdash 5 : \text{int} \rrbracket &= (\text{i64}, \text{Const 5}, [])
\end{align*}
\]

\[
\begin{align*}
\llbracket C \vdash 341 : \text{int} \rrbracket &= (\text{i64}, \text{Const 341}, []) \\
\llbracket C \vdash 5 : \text{int} \rrbracket &= (\text{i64}, \text{Const 5}, [])
\end{align*}
\]

\[
\llbracket C \vdash 341 + 5 : \text{int} \rrbracket = (\text{i64}, \%\text{tmp}, [\%\text{tmp} = \text{add i64 (Const 341) (Const 5)]})
\]
What about the Context?

• What is $\llbracket C \rrbracket$?
• 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?

• $\llbracket C \rrbracket$ maps source identifiers, “$x$” to source types and $\llbracket x \rrbracket$

• What is the interpretation of a variable $\llbracket x \rrbracket$ at the target level?
  – How are the variables used in the type system?

\[
\frac{x : t \in L}{G ; L \vdash x : t} \quad \text{TYP\_VAR} \quad \frac{x : t \in L}{G ; L \vdash \text{exp} : t} \quad \text{TYP\_ASSN}
\]

as expressions
(which denote values)

as addresses
(which can be assigned)
Interpretation of Contexts

• \([C] = \) a map from source identifiers to types and target identifiers

• INARIANT:
  \(x:t \in C\) 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:

\[
\begin{align*}
\text{TYP}_\text{VAR} & \quad \frac{x : t \in L \quad G; L \vdash x : t}{G; L \vdash x \quad \text{as expressions}} \\
& \quad \text{as expressions (which denote values)}
\end{align*}
\]

\[
\begin{align*}
& = (\%\text{tmp}, [\%\text{tmp} = \text{load } i64* \%\text{id}_x]) \\
& \quad \text{where } (i64, \%\text{id}_x) = \text{lookup } [L] x
\end{align*}
\]

- What about statements?

\[
\begin{align*}
\text{TYP}_\text{ASSN} & \quad \frac{x : t \in L \quad G; L \vdash \text{exp} : t}{G; L; rt \vdash x = \text{exp} ; \Rightarrow L} \\
& \quad \text{as addresses (which can be assigned)}
\end{align*}
\]

\[
\begin{align*}
& = \text{stream @} \\
& \quad [\text{store } [t] \text{ opn}, [t]* \%\text{id}_x] \\
& \quad \text{where } (t, \%\text{id}_x) = \text{lookup } [L] x \\
& \quad \text{and } [G; L \vdash \text{exp} : t] = ([t], \text{opn}, \text{stream})
\end{align*}
\]
Other Judgments?

- **Statement:**
  \[
  \llbracket C; rt \vdash \text{stmt} \Rightarrow C' \rrbracket = \llbracket C' \rrbracket, \text{stream}
  \]

- **Declaration:**
  \[
  \llbracket G; L \vdash t\ x = \exp \Rightarrow G; L, x: t \rrbracket = \llbracket G; L, x: t \rrbracket, \text{stream}
  \]

**INVARIANT:** stream is of the form:

\[
\text{stream'} \oplus
\[
[ \ %id_x = \text{alloca} \ [t];
  \text{store} \ [t] \ \text{opn}, \ [t] \ast \ %id_x ]
\]

and \[
\llbracket G; L \vdash \text{exp} : t \rrbracket = (\llbracket t \rrbracket, \text{opn}, \text{stream'})
\]

- **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.

\[
[C;rt \vdash \text{while}(e) \ s \Rightarrow C'] = [C'],
\]

lpre:
  \[
  \text{opn} = [C \vdash e : \text{bool}]
  \%\text{test} = \text{icmp eq i1 opn, 0}
  \text{br} \%\text{test}, \text{label } \%\text{lpost}, \text{label } \%\text{lbody}
  \]

lbody:
  \[
  [C;rt \vdash s \Rightarrow C']
  \text{br} \%\text{lpre}
  \]

lpost:

• Note: writing \( \text{opn} = [C \vdash e : \text{bool}] \) is pun
  – translating \([C \vdash e : \text{bool}]\) generates code that puts the result into \( \text{opn} \)
  – 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 and the else branch is optional.

\[
[C;rt \vdash \text{if } (e_1) \ s_1 \ \text{else } s_2 \ \Rightarrow C'] = [C']
\]

```plaintext
opn = \([C \vdash e : \text{bool}]\)
%test = icmp eq i1 opn, 0
br %test, label %else, label %then
then:
    \([C;rt \vdash s_1 \Rightarrow C']\)
    br %merge
else:
    \([C; rt s_2 \Rightarrow C']\)
    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
OPTIMIZING CONTROL
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 %tmp4, label %else, label %then

then:
    store [%z], 3
    br %merge

else:
    store [%z], 4
    br %merge

merge:
    %tmp5 = load [%z]
    ret %tmp5
```
Observation

• Usually, we want the translation $⟦e⟧$ to produce a value
  – $⟦C ⊢ e : t⟧ = (ty, operand, stream)$
  – e.g. $⟦C ⊢ e_1 + e_2 : int⟧ = (i64, %tmp, [%tmp = add ⟦e_1⟧ ⟦e_2⟧))$

• But when the expression we’re compiling appears in a test, the program jumps to one label or another after the comparison but otherwise 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:

\[ [C \vdash e : t] = (ty, \text{operand, stream}) \]

Conditional branch translation of booleans, without materializing the value:

\[ [C \vdash e : \text{bool@}] \text{ltrue lfalse} = \text{stream} \]

\[ [C, rt \vdash \text{if (e) then s1 else s2 } \Rightarrow C'] = [C'], \]

\[ \text{insns}_3 \]

then:

\[ [s_1] \]

\[ \text{br %merge} \]

else:

\[ [s_2] \]

\[ \text{br %merge} \]

merge:

Notes:

- takes two extra arguments: a “true” branch label and a “false” branch label.
- Doesn’t “return a value”
- Aside: this is a form of continuation-passing translation…

where

\[ [C, rt \vdash s_1 \Rightarrow C'] = [C'], \text{insns}_1 \]

\[ [C, rt \vdash s_2 \Rightarrow C''] = [C''], \text{insns}_2 \]

\[ [C \vdash e : \text{bool@}] \text{then else} = \text{insns}_3 \]
Short Circuit Compilation: Expressions

- \[ [C \vdash e : \text{bool}] \] $ltrue$ $lfalse = \text{insns}$

\[
[C \vdash \text{false} : \text{bool}] \ ltrue \ lfalse = \br \ %lfalse
\]

\[
[C \vdash \text{true} : \text{bool}] \ ltrue \ lfalse = \br \ %ltrue
\]

\[
[C \vdash e : \text{bool}] \ lfalse \ ltrue = \text{insns}
\]

\[
[C \vdash \neg e : \text{bool}] \ ltrue \ lfalse = \text{insns}
\]

FALSE

TRUE

NOT
Short Circuit Evaluation

Idea: build the logic into the translation

\[
\begin{align*}
[C \vdash e_1 : \text{bool}@] \ ltrue \ right &= \ insns_1 & [C \vdash e_2 : \text{bool}@] \ ltrue \ lfalse &= \ insns_2 \\
[C \vdash e_1 | e_2 : \text{bool}@] \ ltrue \ lfalse &= \ insns_1 \\
& \quad \text{right:} \\
& \quad \text{insn}_2 \\
[C \vdash e_1 \& e_2 : \text{bool}@] \ ltrue \ lfalse &= \ insns_1 \\
& \quad \text{right:} \\
& \quad \text{insn}_2
\end{align*}
\]

where \( right \) is a fresh label
Short-Circuit Evaluation

- Consider compiling the following program fragment:

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

```asm
%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
```
See fun.ml

BACK TO LAMBDA CALCULUS: INTERPRETERS PART II
Compiling lambda calculus to straight-line code.
Representing evaluation environments at runtime.
To implement first-class functions on a processor, there are two problems:

- First: we must implement substitution of free variables
- Second: we 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:
  \[
  \text{let add} = \text{fun } x \rightarrow \text{fun } y \rightarrow x + y
  \]

• 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 code pointer takes a pair of parameters: env and y
  – The function code is (essentially):
    \[
    \text{fun (env, y) } \rightarrow \text{ let } x = \text{nth env 0 } \text{ in } x + y
    \]
Representing Closures

• As we saw, the simple closure conversion algorithm 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 datastructure 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.
Adding Integers to Lambda Calculus

\[
\begin{align*}
\text{exp ::= } & \quad| \ldots \\
& \quad| n \quad \text{constant integers} \\
& \quad| \text{exp}_1 + \text{exp}_2 \quad \text{binary arithmetic operation}
\\
\text{val ::= } & \quad| \text{fun } x \rightarrow \text{exp} \quad \text{functions are values} \\
& \quad| n \quad \text{integers are values}
\\
n \{v/x\} = n \quad \text{constants have no free vars.} \\
(e_1 + e_2) \{v/x\} = (e_1 \{v/x\} + e_2 \{v/x\}) \quad \text{substitute everywhere}
\end{align*}
\]

\[
\begin{align*}
\text{exp}_1 \downarrow n_1 \quad \text{exp}_2 \downarrow n_2 \\
\hline \\
\text{exp}_1 + \text{exp}_2 \downarrow (n_1 \ [+] \ n_2)
\end{align*}
\]

Object-level ‘+’  Meta-level ‘+’