Announcements

• Midterm Exam:
  Location: Meyerson Hall B3
  MEYH  B3
  Thursday, March 5 noon-1:30

• HW4: Oat v. 1 Frontend
  – Due: March 26th at 11:59pm
  – Counts as 2x project
  – Start early!
See lec14.zip

OAT V.0
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*:

<table>
<thead>
<tr>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G;L \vdash e : \text{bool} ) \quad ( G;L;rt \vdash s_1 ) \quad ( G;L;rt \vdash s_2 )</td>
<td>( G;L;rt \vdash \text{if} \ (e) \ s_1 \ \text{else} \ s_2 )</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 oat0-defn.pdf:

```plaintext
int x1 = 0;
int x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

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

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

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

\[ D_1 \cdot D_2 \quad D_3 \quad D_4 \]

\[ G_0; \cdot; \text{int} \vdash \text{int} \quad \text{x}_1 \quad \text{int} \quad \text{x}_2 \quad \text{int} \]

\[ \vdash \text{int} \quad \text{x}_1 = 0; \quad \text{int} \quad \text{x}_2 = \text{x}_1 + \text{x}_1; \quad \text{x}_1 = \text{x}_1 - \text{x}_2; \quad \text{return} \quad \text{x}_1; \quad \Rightarrow \quad \cdot; \text{x}_1; \text{int}, \text{x}_2; \text{int} \]

\[ \text{[STMTS]} \quad \text{[PROG]} \]
Example Derivation

\[ D_1 = \frac{G_0; \vdash 0 : \text{int}}{\text{INT}} \]

\[ G_0; \vdash 0 : \text{int} \quad \text{[CONST]} \]

\[ G_0; \vdash \text{int } x_1 = 0 \Rightarrow \cdot , x_1 : \text{int} \quad \text{[DECL]} \]

\[ G_0; \cdot ; \text{int} \vdash \text{int } x_1 = 0; \Rightarrow \cdot , x_1 : \text{int} \quad \text{[SDECL]} \]

\[ D_2 = \frac{\vdash + : (\text{int, int}) \to \text{int}}{\text{ADD}} \]

\[ x_1 : \text{int} \in \cdot , x_1 : \text{int} \quad \text{[VAR]} \]

\[ G_0; \cdot , x_1 : \text{int} \vdash x_1 : \text{int} \quad \text{[VAR]} \]

\[ G_0; \cdot , x_1 : \text{int} \vdash x_1 + x_1 : \text{int} \quad \text{[BOP]} \]

\[ G_0; \cdot , x_1 : \text{int}; \text{int} \vdash \text{int } x_2 = x_1 + x_1; \Rightarrow \cdot , x_1 : \text{int}, x_2 : \text{int} \quad \text{[DECL]} \]

\[ G_0; \cdot , x_1 : \text{int}; \text{int} \vdash \text{int } x_2 = x_1 + x_1; \Rightarrow \cdot , x_1 : \text{int}, x_2 : \text{int} \quad \text{[SDECL]} \]
Example Derivation

\[ D_3 = \begin{align*}
& \vdash -(\text{int,int}) \to \text{int} \\
& \begin{array}{c}
\vdash \ \vdash \ x_1 : \text{int} \in \cdot, x_1 : \text{int}, x_2 : \text{int} \\
\vdash \ G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int} \\
\vdash \ G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 - x_2 : \text{int} \\
\vdash \ G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}
\end{array}
\end{align*} \]

\[ D_4 = \begin{align*}
& \vdash \cdot, x_1 : \text{int}, x_2 : \text{int} \\
& \begin{array}{c}
\vdash \ G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash x_1 : \text{int} \\
\vdash \ G_0 ; \cdot, x_1 : \text{int}, x_2 : \text{int} \vdash \text{return } x_1; \Rightarrow \cdot, x_1 : \text{int}, x_2 : \text{int}
\end{array}
\end{align*} \]
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*:
  
  \[ \mathcal{C} \vdash e : t \]
  
  – 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.
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 \[ \llbracket C \vdash e : t \rrbracket = ty, \text{ operand, stream} \]
then the type (at the target level) of the operand is \[ ty = \llbracket t \rrbracket \]
Example

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

$\llbracket \vdash 341 : \text{int} \rrbracket = (\text{i64}, \text{Const 341}, [])$
$\llbracket \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const 5}, [])$

----------------------------------------
$\llbracket C \vdash 341 : \text{int} \rrbracket = (\text{i64}, \text{Const 341}, [])$
$\llbracket C \vdash 5 : \text{int} \rrbracket = (\text{i64}, \text{Const 5}, [])$

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

- What is \([C]\)?
- Source level C has bindings like: \(x: \text{int}, y: \text{bool}\)
  - 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?

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

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

- INVARIANT:
  \(x:t \in C\) means that
  
  1. \(\text{lookup } [C] x = (t, \%id_x)\)
  2. the (target) type of \(\%id_x\) is \([t]^*\) (a pointer to \([t]\))
Interpretation of Variables

• Establish invariant for expressions:

\[
\begin{align*}
&\frac{x : t \in L}{G ; L \vdash x : t} \\
&\text{TYP\_VAR}
\end{align*}
\]

as expressions (which denote values)

\[
\begin{align*}
&\frac{x : t \in L}{G ; L \vdash x = exp \Rightarrow L} \\
&\text{TYP\_ASSN}
\end{align*}
\]

as addresses (which can be assigned)

\[
\begin{align*}
&\frac{x : t \in L}{G ; L \vdash \text{return} \Rightarrow L} \\
&\text{stmt}
\end{align*}
\]

where \((\text{int}, \%id_x) = \text{lookup} [L] x\)

• What about statements?

\[
\begin{align*}
&\frac{x : t \in L}{G ; L \vdash \text{block} \Rightarrow L} \\
&\text{stmt}
\end{align*}
\]

where \((\text{int}, \%id_x) = \text{lookup} [L] x\)

and \([G;L \vdash \text{exp} : t] = ([t], \text{opn}, \text{stream})\)
Other Judgments?

• Statement:
\[ C; \text{rt} \vdash \text{stmt} \Rightarrow C' \] = \[ C' \], stream

• Declaration:
\[ G;L \vdash t \ x = \text{exp} \Rightarrow G;L,x:t \] = \[ G;L,x:t \], stream

INVARIANT: stream is of the form:
\[
\text{stream}' @ \\
[ \%id_x = \text{alloca } [t]; \\
\quad \text{store } [t] \ \text{opn}, [t]^* \ %id_x ]
\]

and \[ G;L \vdash \text{exp} : t \] = (\[ t \], 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.

\[ [C;rt \downarrow \text{while}(e) \rightarrow C'] = [C'] , \]

\[
\text{lpre:}
\begin{align*}
&\text{opn} = [C \vdash e : \text{bool}] \\
&\text{%test} = \text{icmp eq il opn, 0} \\
&\text{br %test, label %lpost, label %lbody}
\end{align*}
\]

\[
\text{lbody:}
\begin{align*}
&[C;rt \vdash s \Rightarrow C'] \\
&\text{br %lpre}
\end{align*}
\]

\[
\text{lpost:}
\]

- 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.

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

```plaintext
opn = [C \vdash e : bool]
%test = icmp eq il 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 $\llbracket e \rrbracket$ to produce a value
  – $\llbracket C \vdash e : t \rrbracket = (\text{ty}, \text{operand, stream})$
  – e.g. $\llbracket C \vdash e_1 + e_2 : \text{int} \rrbracket = (\text{i64}, \%\text{tmp}, \%\text{tmp} = \text{add} \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket)$

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

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

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

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

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

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

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

\[ \left[ C \vdash !e : \text{bool@} \right] \text{ltrue lfalse} = \text{insns} \]
Short Circuit Evaluation

Idea: build the logic into the translation

\[
\begin{align*}
\llbracket C \vdash e_1 : \text{bool}@ \rrbracket \ & \ ltrue \ \text{right} = \ \text{insns}_1 \quad \& \quad \llbracket C \vdash e_2 : \text{bool}@ \rrbracket \ & \ ltrue \ \text{lfalse} = \ \text{insns}_2 \\
\llbracket C \vdash e_1 \ | e_2 : \text{bool}@ \rrbracket \ & \ ltrue \ \text{lfalse} = \\
\llbracket C \vdash e_1 \& e_2 : \text{bool}@ \rrbracket \ & \ ltrue \ \text{lfalse} = \\
\end{align*}
\]

where \text{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
```