Review Type Judgments

```
int program(int argc, string[] argv) {
  if (true) {
    return 0;
  } else {
    return 1;
  }
}
```

$$G; L_0 \vdash vdecls \Rightarrow L_i$$

$$G; L_1; retty \vdash stmt \Rightarrow L_2$$

$$G; L_0; t \vdash stmt_1 .. stmt_i \Rightarrow L_i$$

$$G; L_1 \vdash vdecl \Rightarrow L_2$$

$$G \vdash decl$$

$$\vdash prog$$

$$G_0 \vdash decl \Rightarrow G_1$$

$$G; L; t \vdash block$$

Review Type Judgments

```
int program(int argc, string[] argv) {
  while (1 > 0) {
    return 3;
  }
  return 0;
}
```

Review Type Judgments

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

int f(A x) {
  return 3;
}

int g(B x) {
  return 3;
}

int l() {
  var x = g;
  x = f;
  return 3;
}
```

CS 516: COMPILERS

Lecture 16

Topics

- Review Type Derivations
- Compilation as translating Judgments

COMPILATION AS TRANSLATING JUDGMENTS

Why Inference Rules?

- They are a compact, precise way of specifying language properties.
- Correspond closely to recursive AST traversal
- Strong mathematical foundations
- Type checking - search backward through the rules
- It works for **compiling**, too!

- Compiling in a context is
  - Compilation follows the typechecking judgment
  - nothing more an "interpretation" of the inference rules that specify typechecking*: $[\![C \vdash e : t]\!]$

*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?

$$[\![C \vdash e : t]\!] = ?$$

  - $[\![C]\!]$ translates contexts
  - $[\![t]\!]$ is a target type

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?

$$[\![C \vdash e : t]\!] = ?$$

  - $[\![C]\!]$ translates contexts
  - $[\![t]\!]$ is a target type
  - $[\![e]\!]$ translates to a (potentially empty) stream of instructions, that, when run, computes the result into some operand

- ***Invariant***: if $[\![C \vdash e : t]\!] =$ ***ty, operand , stream***
  then the type (at the target level) of the operand is ty=$[\![t]\!]$

Example

- $C \vdash 516 + 5 : \text{int}$ what is $[\![C \vdash 516 + 5 : \text{int}]\!]$?

$$\dfrac{\dfrac{\vdash 516 : \text{int}}{C \vdash 516 : \text{int}} \qquad \dfrac{\vdash 5 : \text{int}}{C \vdash 5 : \text{int}}}{C \vdash 516 + 5 : \text{int}}$$

Example

- $C \vdash 516 + 5 : \text{int}$ what is $[\![C \vdash 516 + 5 : \text{int}]\!]$?

$[\![\vdash 516 : \text{int}]\!] =$ $[\![\vdash 5 : \text{int}]\!] =$

$[\![C \vdash 516 : \text{int}]\!] =$ $[\![C \vdash 5 : \text{int}]\!] =$

$[\![C \vdash 516 + 5 : \text{int}]\!] =$

Example

- C ⊢ 516 + 5 : int what is ⟦ C ⊢ 516 + 5 : int⟧ ?

Invariant: $[\![C \vdash e : t]\!]$ = ***ty, operand , stream***

⟦ ⊢ 516 : int ⟧ = ??? ⟦⊢ 5 : int⟧ = ???

⟦C ⊢ 516 : int⟧ = ??? ⟦C ⊢ 5 : int⟧ = ???

⟦C ⊢ 516 + 5 : int⟧ = ???

Example

- $C \vdash 516 + 5 : \text{int}$ what is $[\![C \vdash 516 + 5 : \text{int}]\!]$?

Invariant: $[\![C \vdash e : t]\!]$ = ***ty, operand , stream***

$[\![\vdash 516 : \text{int}]\!]$ = `(i64, Const 516, [])`

$[\![\vdash 5 : \text{int}]\!]$ = ???

$[\![C \vdash 516 : \text{int}]\!]$ = ???

$[\![C \vdash 5 : \text{int}]\!]$ = ???

$[\![C \vdash 516 + 5 : \text{int}]\!]$ = ???

Example

- C ⊢ 516 + 5 : int what is ⟦ C ⊢ 516 + 5 : int⟧ ?

Invariant: $[\![C \vdash e : t]\!]$ = *ty, operand , stream*

⟦ ⊢ 516 : int ⟧ = `(i64, Const 516, [])`

⟦⊢ 5 : int⟧ = ???

⟦C ⊢ 516 : int⟧ = `(i64, Const 516, [])`

⟦C ⊢ 5 : int⟧ = ???

⟦C ⊢ 516 + 5 : int⟧ = ???

Example

- C ⊢ 516 + 5 : int what is ⟦ C ⊢ 516 + 5 : int⟧ ?

Invariant: $[\![C \vdash e : t]\!]$ = *ty, operand , stream*

⟦ ⊢ 516 : int ⟧ = `(i64, Const 516, [])`

⟦⊢ 5 : int⟧ = `(i64, Const 5, [])`

⟦C ⊢ 516 : int⟧ = `(i64, Const 516, [])`

⟦C ⊢ 5 : int⟧ = ???

⟦C ⊢ 516 + 5 : int⟧ = ???

Example

- C ⊢ 516 + 5 : int what is ⟦ C ⊢ 516 + 5 : int⟧ ?

Invariant: $[\![C \vdash e : t]\!]$ = *ty, operand , stream*

$[\![\vdash 516 : \text{int}]\!]$ = `(i64, Const 516, [])`

$[\![C \vdash 516 : \text{int}]\!]$ = `(i64, Const 516, [])`

$[\![\vdash 5 : \text{int}]\!]$ = `(i64, Const 5, [])`

$[\![C \vdash 5 : \text{int}]\!]$ = `(i64, Const 5, [])`

$[\![C \vdash 516 + 5 : \text{int}]\!]$ = ???

Example

- $C \vdash 516 + 5 : \text{int}$ what is $[\![C \vdash 516 + 5 : \text{int}]\!]$?

Invariant: $[\![C \vdash e : t]\!]$ = *ty, operand , stream*

$[\![\vdash 516 : \text{int}]\!]$ = `(i64, Const 516, [])` $[\![\vdash 5 : \text{int}]\!]$ = `(i64, Const 5, [])`

$[\![C \vdash 516 : \text{int}]\!]$ = `(i64, Const 516, [])` $[\![C \vdash 5 : \text{int}]\!]$ = `(i64, Const 5, [])`

$[\![C \vdash 516 + 5 : \text{int}]\!]$ = `(i64, %tmp, [%tmp = add i64 (Const 516) (Const 5)])`

What about the Context?

Question: What is ⟦C⟧?

- 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?

- ⟦C⟧ maps *source identifiers*, "x" to source types and ⟦x⟧

What about the Context?

Question: What is ⟦C⟧?

- 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?

- ⟦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? *Two ways*:

What about the Context?

Question: What is ⟦C⟧?

- 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?

- ⟦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? *Two ways*:

$$\frac{x : t \in L}{G; L \vdash x : t}\ \text{TYP\_VAR}$$

What about the Context?

Question: What is ⟦C⟧?

- 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?

- ⟦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? *Two ways*:

As exprs (Values)

$$\frac{x : t \in L}{G; L \vdash x : t} \text{ TYP\_VAR}$$

As addrs (Can be assigned)

$$\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L} \text{ TYP\_ASSN}$$

Interpretation of Contexts

- **How to interpret contexts:**

  ⟦C⟧ = a map from source identifiers to types and target identifiers

- **Invariant**:

  $x : t \in C$ means that

  (1) lookup ⟦C⟧ x = (t, `%id_x`)

  (2) the (target) type of `%id_x` is ⟦t⟧* (a pointer to ⟦t⟧)

Where to get x in the target code

The type of that thing in the target notion of types

Interpretation of Variables

- **Expressions:** Invariant has the following form:

$$\frac{x : t \in L}{G; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

Translation of the **expression** x

```
= (%tmp, [%tmp = load i64* %id_x])
```

where (`i64`, `%id_x`) = lookup $[\![L]\!]$ x

Translation tells you the target type…

… and where it's stored

Interpretation of Variables

- **Expressions:** Invariant has the following form:

$$\frac{x : t \in L}{G; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

Translation of the **expression** x

```
= (%tmp, [%tmp = load i64* %id_x])
```

where (`i64`, `%id_x`) = lookup $[\![L]\!]$ x

Translation tells you the target type…

… and where it's stored

Interpretation of Variables

- **Expressions:** Invariant has the following form:

$$\frac{x : t \in L}{G; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

Translation of the **expression** x

= `(%tmp, [%tmp = load i64* %id_x])`

where (`i64`, `%id_x`) = lookup $[\![L]\!]$ x

Translation tells you the target type…

… and where it's stored

- **Statements:** Invariant has the following form:

$$\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L}\ \text{TYP\_ASSN}$$

as addresses
(which can be assigned)

= stream @
`[store [[t]] opn, [[t]]* %id_x]`

Interpretation of Variables

- **Expressions:** Invariant has the following form:

$$\frac{x : t \in L}{G; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

Translation of the **expression** x

= `(%tmp, [%tmp = load i64* %id_x])`

where (`i64`, `%id_x`) = lookup $[\![L]\!]$ x

Translation tells you the target type…

… and where it's stored

- **Statements:** Invariant has the following form:

$$\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L}\ \text{TYP\_ASSN}$$

as addresses
(which can be assigned)

= stream @
`[store [[t]] opn, [[t]]* %id_x]`

where (t, `%id_x`) = lookup $[\![L]\!]$ x

and $[\![G; L \vdash exp : t]\!]$ = ($[\![t]\!]$, `opn`, stream)

1. First recursively translate *exp* to find out where to store

Interpretation of Variables

- **Expressions:** Invariant has the following form:

$$\frac{x : t \in L}{G; L \vdash x : t}\ \text{TYP\_VAR}$$

as expressions
(which denote values)

Translation of the **expression** x

= `(%tmp, [%tmp = load i64* %id_x])`

where (`i64`, `%id_x`) = lookup $[\![L]\!]$ x

Translation tells you the target type…

… and where it's stored

- **Statements:** Invariant has the following form:

$$\frac{x : t \in L \quad G; L \vdash exp : t}{G; L; rt \vdash x = exp; \Rightarrow L}\ \text{TYP\_ASSN}$$

as addresses
(which can be assigned)

2. Then create a new store instruction:

= stream @
`[store [[t]] opn, [[t]]* %id_x]`

where (t, `%id_x`) = lookup $[\![L]\!]$ x

and $[\![G; L \vdash exp : t]\!]$ = ($[\![t]\!]$, `opn`, stream)

1. First recursively translate *exp* to find out where to store

Other Judgments?

- **Statement**: How to compile stmt typing rules

  $[\![C; rt \vdash stmt \Rightarrow C']\!] = \quad [\![C']\!]$, stream

  stmt may alter context

  may need **C'** to get values from **stream**

- **Declaration**: How to compile decl rules
-

  $[\![G; L \vdash t\ x = exp \Rightarrow G; L, x : t]\!] = \quad [\![G; L, x : t]\!]$, stream

  INVARIANT: stream is of the form:

```
stream' @
[ %id_x = alloca ⟦t⟧;
  store ⟦t⟧ opn, ⟦t⟧* %id_x ]
```

  and $[\![G; L \vdash exp : t]\!] = ([\![t]\!],$ `opn`, `stream'`$)$

- Rest follow similarly

Other Judgments?

- **Statement**: How to compile stmt typing rules

  $[\![C; rt \vdash stmt \Rightarrow C']\!] = \quad [\![C']\!]$, stream

  stmt may alter context

  may need **C'** to get values from **stream**

- **Declaration**: How to compile decl rules
-

  $[\![G; L \vdash t\ x = exp \Rightarrow G; L, x : t]\!] = \quad [\![G; L, x : t]\!]$, stream

  extended context with new typing

  INVARIANT: stream is of the form:

```
stream' @
[ %id_x = alloca [[t]];
  store [[t]] opn, [[t]]* %id_x ]
```

  and $[\![G; L \vdash exp : t]\!] = ([\![t]\!], \texttt{opn}, \texttt{stream'})$

- Rest follow similarly

Other Judgments?

- **Statement**: How to compile stmt typing rules

  $[\![C; rt \vdash stmt \Rightarrow C']\!] = \quad [\![C']\!]$, stream

  stmt may alter context

  may need **C'** to get values from **stream**

- **Declaration**: How to compile decl rules
-

  $[\![G; L \vdash t\ x = exp \Rightarrow G; L, x : t]\!] = \quad [\![G; L, x : t]\!]$, stream

  extended context with new typing

  INVARIANT: stream is of the form:

```
stream' @
[ %id_x = alloca [[t]];
   store [[t]] opn, [[t]]* %id_x ]
```

  allocate space

  store initial value

  and $[\![G; L \vdash exp : t]\!] = ([\![t]\!], \texttt{opn}, \texttt{stream'})$

- Rest follow similarly

Homework 5

```
let rec cmp_exp (tc : TypeCtxt.t) (c:Ctxt.t) (exp:Ast.exp node) :
Ll.ty * Ll.operand * stream =

  match exp.elt with
  | Ast.CInt i  -> I64, Const i, []
  | Ast.CNull r -> cmp_ty tc (TNullRef r), Null, []

  | Ast.CStr s ->
    let gid = gensym "str_arr" in
    let str_typ = str_arr_ty s in
    let uid = gensym "str" in
        Ptr I8, Id uid, []
    >:: G(gid, (str_typ, GString s))
    >:: I(uid, Gep(Ptr str_typ, Gid gid, [Const 0L; Const 0L;]))
```

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.

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

```
lpre:
    opn = [[C ⊢ e : bool]]
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
    [[C; rt ⊢ s ⇒ C']]
    br %lpre
lpost:
```

- Note: writing `opn` $= [\![C \vdash e : \texttt{bool}]\!]$ is pun
  - translating $[\![C \vdash e : \text{bool}]\!]$ generates *code* that puts the result into `opn`
  - In this notation there is implicit collection of the code

Translating while

- Consider translating "`while(e) s`":
  - Test the conditional, if true jump to the body, else jump to the label after the body.

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

```
lpre:
    opn = [[C ⊢ e : bool]]
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
    [[C; rt ⊢ s ⇒ C']]
    br %lpre
lpost:
```

- Note: writing `opn` $= [\![C \vdash e : \texttt{bool}]\!]$ is pun
  - translating $[\![C \vdash e : \texttt{bool}]\!]$ generates code that puts the result into `opn`
  - In this notation

$$\frac{H; G; L \vdash exp : \texttt{bool} \qquad H; G; L; rt \vdash block; r}{H; G; L; rt \vdash \texttt{while}(exp)\ block \Rightarrow L; \bot}$$

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.

$[\![C; rt \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \Rightarrow C']\!] = [\![C']\!]$

```
    opn = [[C ⊢ e : bool]]
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
    [[C; rt ⊢ s1 ⇒ C']]
    br %merge
else:
    [[C; rt ⊢ s2 ⇒ C']]
    br %merge
merge:
```

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.

$[\![C; rt \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \Rightarrow C']\!] = [\![C']\!]$

```
    opn = [[C ⊢ e : bool]]
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
    [[C; rt ⊢ s1 ⇒ C']]
    br %merge
else:
    [[C; rt ⊢ s2 ⇒ C']]
    br %merge
merge:
```

$$\frac{H;G;L \vdash exp : \texttt{bool} \quad H;G;L;rt \vdash block_1;r_1 \quad H;G;L;rt \vdash block_2;r_2}{H;G;L;rt \vdash \texttt{if}(exp)\ block_1\ \texttt{else}\ block_2 \Rightarrow L; r_1 \wedge r_2}$$

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,
  for example, function `cmp_stmt` case `If(g,st1,st2)`

OPTIMIZING CONTROL

Standard Evaluation

- Consider compiling the following program fragment:

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

```
    %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
  - $⟦C \vdash e : t⟧$ = (ty, operand, stream)
  - e.g. $⟦C \vdash e_1 + e_2 : int⟧$ = (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, operand, stream)

Conditional branch translation:

Translate booleans, without materializing the value:

$[\![C \vdash e$: bool@$]\!]$ ltrue lfalse = stream

No ty or operand. Instead a stream that uses true and lfalse

$[\![C, rt \vdash \text{if } (e) \text{ then } s_1 \text{ else } s_2 \Rightarrow C']\!] = [\![C']\!]$,

```
    insns3
then:
    [[s1]]
    br %merge
else:
    [[s2]]
    br %merge
merge:
```

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!]$, $insns_1$

$[\![C, rt \vdash s_2 \Rightarrow C'']\!] = [\![C'']\!]$, $insns_2$

$[\![C \vdash e$: bool@ $]\!]$ `then else` = $insns_3$

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…

Short Circuit Compilation: Expressions

- ⟦C ⊢ e : bool@⟧ ltrue lfalse = insns

$$\frac{}{\llbracket C \vdash \texttt{false} : \texttt{bool@} \rrbracket\ l_{true}\ l_{false} = \texttt{[br \%lfalse]}}\ \text{FALSE}$$

$$\frac{}{\llbracket C \vdash \texttt{true} : \texttt{bool@} \rrbracket\ l_{true}\ l_{false} = \texttt{[br \%ltrue]}}\ \text{TRUE}$$

$$\frac{\llbracket C \vdash e : \texttt{bool@} \rrbracket\ l_{false}\ l_{true} = insns}{\llbracket C \vdash \texttt{!}e : \texttt{bool@} \rrbracket\ l_{true}\ l_{false} = insns}\ \text{NOT}$$

Short-Circuit Evaluation

Idea: build the logic into the translation

$$\frac{[\![C \vdash e1 : bool@]\!]\ ltrue\ right = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1 \,|\, e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

$$\frac{[\![C \vdash e1 : bool@]\!]\ right\ lfalse = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1\,\&\,e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

where `right` is a fresh label

Short-Circuit Evaluation

- Consider compiling the following program fragment:

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

```
    %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
```