Lecture 18

EECS 483: COMPILER CONSTRUCTION

Announcements

- HW4: OAT v. 1.0
  - Due tomorrow evening.
- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Tuesday, April 9th
  - Available late tomorrow, review during Wednesday's lecture.
  - **As usual, Start Early!**
- Next two weeks
  - Professor New will be on family starting soon (at the latest next Monday)
    - Lectures by Eric or guest professors
    - Only remote office hours for Prof. New, no appointments
  - April 8: Total solar eclipse (only one this century in the USA!)
    - During lecture time, so we will cancel.

Type Judgments

- In the judgment: $E \vdash e : t$
  - E is a ***typing environment*** or a ***type context***
  - E maps variables to types. It is just a set of bindings of the form: $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$

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

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

$$\frac{}{E \vdash i : int}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : int \qquad E \vdash e_2 : int}{E \vdash e_1 + e_2 : int}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \ : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1 \ e_2 : S}$$

- Note how these rules correspond to the code.

Type 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 typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:

$$\vdash \text{(fun (x:int) -> x + 3) 5 : int}$$

Example Derivation Tree

$$\text{APP}\ \dfrac{\text{FUN}\ \dfrac{\text{ADD}\ \dfrac{\text{VAR}\ \dfrac{x : \text{int} \ \in \ x : \text{int}}{x : \text{int} \vdash x \ : \text{int}} \qquad \text{INT}\ \dfrac{}{x : \text{int} \vdash 3 \ : \text{int}}}{x : \text{int} \vdash x + 3 : \text{int}}}{\vdash (\text{fun } (x\text{:int}) \rightarrow x + 3) : \text{int} \longrightarrow \text{int}} \qquad \text{INT}\ \dfrac{}{\vdash 5 : \text{int}}}{\vdash (\text{fun } (x\text{:int}) \rightarrow x + 3)\ 5 \ : \text{int}}$$

- Note: the OCaml function **typecheck** verifies the existence of this tree. The structure of the recursive calls when running **typecheck** is the same shape as this tree!
- Note that $x : \text{int} \ \in \ E$ is implemented by the function **lookup**

Ill-typed Programs

- Programs without derivations are ill-typed

Example: There is no type T such that

$\vdash$ (fun (x:int) → x 3) 5 : T

$x : int \rightarrow T \notin x : int$

VAR

$x : int \vdash x : int \rightarrow T$ $\quad$ $x : int \vdash 3 : int$

APP

$x : int \vdash x\ 3 : T$

FUN

$\vdash$ (fun (x:int) →x 3) : $int \rightarrow T$ $\quad$ $\vdash 5 : int$

APP

$\vdash$ (fun (x:int) → x 3) 5 : T

Type Safety

"Well typed programs do not go wrong."

– Robin Milner, 1978

Theorem: (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$.

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

Notes about this Typechecker

- The interpreter evaluates the body of a function only when it's applied.
- The typechecker always checks the body of the function
  - even if it's never applied
  - We *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 -> t_2).

- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get.
  - But we can calculate e_1's type, check that e_2 is an argument of the right type, and determine what type e_1 will return.

- Question: Why is this an approximation?
- Question: What if `well_typed` always returns `false`?

oat.pdf

TYPECHECKING OAT

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.pdf:

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

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

```
var x2 = x1 + x1;
return(x2);
```

Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0\,;\,\cdot\,;\texttt{int} \vdash \texttt{var}\ x_1 \texttt{ = } 0\texttt{; var}\ x_2 \texttt{ = } x_1 + x_1\texttt{;}\ x_1 \texttt{ = } x_1 \texttt{ - } x_2\texttt{; return}\ x_1\texttt{;} \Rightarrow \cdot, x_1\texttt{:int}, x_2\texttt{:int}}\ [\textsc{stmts}]}{\vdash \texttt{var}\ x_1 \texttt{ = } 0\texttt{; var}\ x_2 \texttt{ = } x_1 + x_1\texttt{;}\ x_1 \texttt{ = } x_1 \texttt{ - } x_2\texttt{; return}\ x_1\texttt{;}}\ [\textsc{prog}]$$

Example Derivation

$$\mathcal{D}_1 \quad = \quad \cfrac{\cfrac{\cfrac{\cfrac{}{G_0\,;\cdot \vdash 0 : \texttt{int}}\,[\textsc{int}]}{G_0\,;\cdot \vdash 0 : \texttt{int}}\,[\textsc{const}]}{G_0\,;\cdot \vdash \texttt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \texttt{int}}\,[\textsc{decl}]}{G_0\,;\cdot\,;\texttt{int} \vdash \texttt{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \texttt{int}}\,[\textsc{sdecl}]$$

$$\mathcal{D}_2 \quad = \quad \cfrac{\cfrac{\cfrac{\cfrac{}{\vdash + : (\texttt{int},\texttt{int}) \to \texttt{int}}\,[\textsc{add}] \quad \cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int} \vdash x_1 : \texttt{int}}\,[\textsc{var}] \quad \cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int} \vdash x_1 : \texttt{int}}\,[\textsc{var}]}{G_0\,;\cdot, x_1 : \texttt{int} \vdash x_1 + x_1 : \texttt{int}}\,[\textsc{bop}]}{G_0\,;\cdot, x_1 : \texttt{int}\,;\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\,[\textsc{decl}]}{G_0\,;\cdot, x_1 : \texttt{int}\,;\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\,[\textsc{sdecl}]$$

Example Derivation

$$\mathcal{D}_3 \quad = \quad \cfrac{\cfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \to \texttt{int}}[\text{ADD}] \quad \cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\text{VAR}] \quad \cfrac{x_2 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_2 : \texttt{int}}[\text{VAR}]}{\cfrac{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\text{ASSN}]}[\text{BOP}]$$

$$\mathcal{D}_4 \quad = \quad \cfrac{\cfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\text{VAR}]}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash \texttt{return } x_1 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\text{RET}]$$

Type Safety For General Languages

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 behaviors:
  - 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…

Beyond describing “structure”… describing “properties”

Types as sets

Subsumption

TYPES, MORE GENERALLY

What are types, anyway?

- A *type* is can often be thought of as a predicate on the set of values in a system.
  - For example, 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 taken to be 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:

P-INT
$$\frac{i > 0}{G \vdash i : \text{Pos}}$$

N-INT
$$\frac{i < 0}{G \vdash i : \text{Neg}}$$

ZERO
$$\frac{}{G \vdash 0 : \text{Zero}}$$

- Same for booleans:

TRUE
$$\frac{}{G \vdash \text{true} : \text{True}}$$

FALSE
$$\frac{}{G \vdash \text{false} : \text{False}}$$

What about "if"?

- Two cases are easy:

$$\text{IF-T}\quad \frac{G \vdash e_1 : \text{True} \quad G \vdash e_2 : T}{G \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

$$\text{IF-F}\quad \frac{G \vdash e_1 : \text{False} \quad E \vdash e_3 : T}{G \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

- What happens when we don't know statically which branch will be taken?
- Consider the typechecking problem:

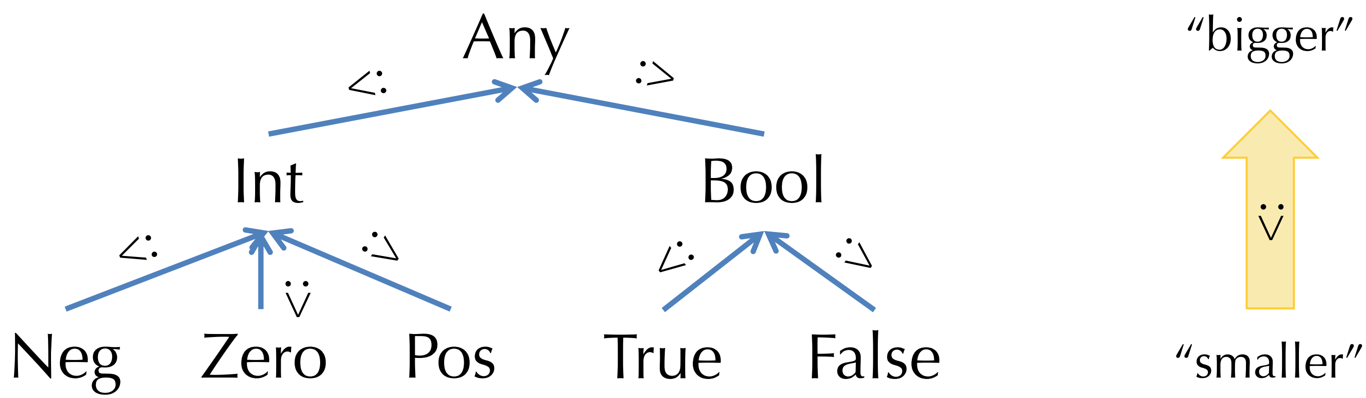
$$x\text{:bool} \vdash \text{if } (x)\ 3 \text{ else } -1 : ?$$

The true branch has type Pos and the false branch has type Neg.

- What should be the result type of the whole if?

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: Pos $\subseteq$ Int
- This subset relation gives rise to a *subtype* relation: Pos <: Int
- Such inclusions give rise to a *subtyping hierarchy*:



- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB) according to the hierarchy.
  - Definition: LUB(T_1, T_2) is the smallest T such that T_1 <: T and T_2 <: T
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
- Note: might want to add types for "NonZero", "NonNegative", and "NonPositive" so that set union on values corresponds to taking LUBs on types.

"If" Typing Rule Revisited

- For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

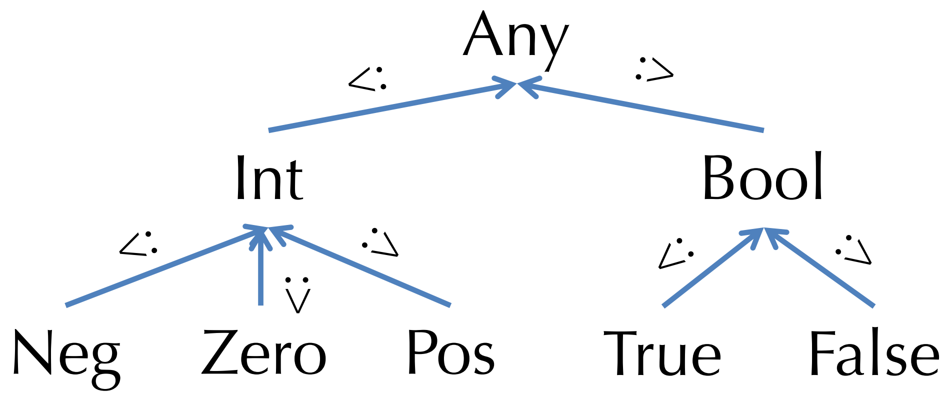
IF-BOOL

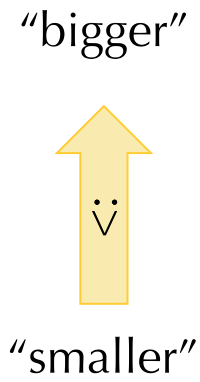
$$\frac{G \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \qquad G \vdash e_3 : T_2}{G \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \text{LUB}(T_1, T_2)}$$

- Note: LUB(T_1, T_2) is the most precise type (according to the hierarchy) that can describe any value that has either type T_1 or type T_2.
- In math notation, LUB(T_1, T_2) is sometimes written $T_1 \vee T_2$
- LUB is also called the *join* operation.

Subtyping Hierarchy

- A *subtyping hierarchy*:





- 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: It $T_1 <: T_2$ 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 *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.
  - *e.g.*, Pos <: Int is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  - *e.g.*, Int <: Pos is *not* sound, since it is *not* the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$

Soundness of LUBs

- Whenever you have a sound subtyping relation, it follows that:
  $$[\![LUB(T_1, T_2)]\!] \supseteq [\![T_1]\!] \cup [\![T_2]\!]$$
  – Note that the LUB is an over approximation of the "semantic union"
  – Example: $[\![LUB(Zero, Pos)]\!] = [\![Int]\!] = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq$
  $\{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\![Zero]\!] \cup [\![Pos]\!]$

- 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 these specific subtypes of Int are *sound* for +

ADD

$$\frac{G \vdash e_1 : T_1 \qquad G \vdash e_2 : T_2 \qquad T_1 <: Int \quad T_2 <: Int}{G \vdash e_1 + e_2 : T_1 \vee T_2}$$