Lecture 17

CS 131: COMPILERS

Announcements

- Midterm: graded by November 28th

Scope, Types, and Context

STATIC ANALYSIS

Variable Scoping

- Consider the problem of determining whether a programmer-declared variable is in scope.
- Issues:
  - Which variables are available at a given point in the program?
  - Shadowing – is it permissible to re-use the same identifier, or is it an error?

- Example: The following program is syntactically correct but not well-formed. (y and q are used without being defined anywhere)

```
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?

Inference Rules

- We can read a judgment $G \vdash e$ as
  "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements s_1, s_2.

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

  holds if $G \vdash e$ and $G \vdash s_1$ and $G \vdash s_2$ all hold.
- More succinctly: we summarize these constraints as an ***inference rule***:

Premises

Conclusion

$$\frac{G \vdash e \qquad G \vdash s_1 \qquad G \vdash s_2}{G \vdash \text{if } (e)\ s_1 \text{ else } s_2}$$

- Such a rule can be used for *any* substitution of the syntactic metavariables G, e, s_1 and s_2.

Scope-Checking Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Given: G, a *set* of variable identifiers, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e$ "the free variables of e are included in G"

$$\frac{x \in G}{G \vdash x}$$

"the variable x is free, but in scope"

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$

"G contains the free variables of e_1 and e_2"

$$\frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \rightarrow e}$$

"x is available in the function body e"

Scope-checking Code

- Compare the OCaml code to the inference rules:
  - structural recursion over syntax
  - the check either "succeeds" or "fails"

```
let rec scope_check (g:VarSet.t) (e:exp) : unit =
  begin match e with
  | Var x -> if VarSet.member x g then () else failwith (x ^ "not in scope")
  | App(e1, e2) -> ignore (scope_check g e1); scope_check g e2
  | Fun(x, e)   -> scope_check (VarSet.union g (VarSet.singleton x)) e
  end
```

$$\frac{x \in G}{G \vdash x}\ \boxed{\text{VAR}} \qquad \frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}\ \boxed{\text{APP}} \qquad \frac{G \cup \{x\} \vdash e}{G \vdash \text{fun}\ x \rightarrow e}\ \boxed{\text{FUN}}$$

- The inference rules are a *specification* of the intended behavior of this scope checking code.
  - they don't specify the order in which the premises are checked

Judgments

- A ***judgment*** is a (meta-syntactic) notation that *names* a relation among one or more sets.
  - The sets are usually built from object-language syntax elements and other "math" sets (*e.g.*, integers, natural numbers, *etc.*)
  - We usually describe them using metavariables that range over the sets.
  - Often use domain-specific notation to ease reading.
  - The meaning of judgments, *i.e.*, which sets they represent, is defined by (collections of) inference rules

- Example: When we say "G ⊢ e is a judgment where G is a context of variables and e is a term, defined by these [...] inference rules" that is shorthand for this "math speak":

  - Let Var be the set of all (syntactic) variables
  - Let Exp be the set {e | e is a term of the untyped lambda calculus}
  - Let **P**(Var) be the (finite) powerset of variables (set of all finite sets)
  - Define well-scoped ⊆ (**P**(Var), Exp) to be a relation satisfying the properties defined by the associated inference rules [...]
  - Then "G ⊢ e" is notation that means that (G, e) ∈ well-scoped

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*
  - *axiom*: rule with no premises that are judgments
  - Example: the VAR rule is an axiom (it doesn't have any ⊢
- Goal of the static checking algorithm: *verify that such a tree exists*.

Example: we can scope check the following lambda calculus term by finding a derivation tree for it:

(fun x -> fun y -> x y) (fun z -> z)

Example Derivation Tree

$$
\text{APP}\;\dfrac{
\text{FUN}\;\dfrac{
\text{FUN}\;\dfrac{
\text{APP}\;\dfrac{
\text{VAR}\;\dfrac{x \in \{x,y\}}{\{x, y\} \vdash x}
\qquad
\text{VAR}\;\dfrac{y \in \{x,y\}}{\{x, y\} \vdash y}
}{\{x, y\} \vdash x\ y}
}{\{x\} \vdash \text{fun } y \text{ -> } x\ y}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } x\ y)}
\qquad
\text{FUN}\;\dfrac{
\text{VAR}\;\dfrac{z \in \{z\}}{\{z\} \vdash z}
}{\vdash (\text{fun } z \text{ -> } z)}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } x\ y)\ (\text{fun } z \text{ -> } z)}
$$

- Note: the OCaml function scope_check verifies the existence of this tree. The structure of the recursive calls when running scope_check is the same shape as this tree!
- Note that $x \in E$ is implemented by the function VarSet.mem

Example Failed Derivation

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{z \in \{x,y\}}{\{x, y\} \vdash z}\,\cancel{\text{VAR}}
\qquad
\dfrac{y \in \{x,y\}}{\{x, y\} \vdash y}\,\text{VAR}
}{\{x, y\} \vdash z\ y}\,\text{APP}
}{\{x\} \vdash \text{fun } y \text{ -> } z\ y}\,\text{FUN}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } z\ y)}\,\text{FUN}
\qquad
\dfrac{
\dfrac{z \in \{z\}}{\{z\} \vdash z}\,\text{VAR}
}{\{\} \vdash (\text{fun } z \text{ -> } z)}\,\text{FUN}
}{\{\} \vdash (\text{fun } x \text{ -> fun } y \text{ -> } z\ y)\ (\text{fun } z \text{ -> } z)}\,\text{APP}
$$

- This program is *not* well scoped
  - The variable z is not bound in the body of the left function.
  - The typing derivation fails because the VAR rule cannot succeed
  - (The other parts of the derivation are OK, though!)

Uses of the inference rules

- We can do proofs by induction on the structure of the derivation.
- For example:

Lemma: If $G \vdash e$ then $fv(e) \subseteq G$.

Proof.

By induction on the derivation that $G \vdash e$.

$$\frac{x \in G}{G \vdash x}$$

- case: VAR then we have $e = x$ (for some variable x) and $x \in G$. But $fv(e) = fv(x) = \{x\}$, but then $\{x\} \subseteq G$.

$$\frac{G \vdash e_1 \quad G \vdash e_2}{G \vdash e_1\ e_2}$$

- case: APP then we have $e = e_1\ e_2$ (for some $e_1\ e_2$) and, by induction, we have $fv(e_1) \subseteq G$ and $fv(e_2) \subseteq G$, so $fv(e_1\ e_2) = fv(e_1) \cup fv(e_2) \subseteq G$

$$\frac{G \cup \{x\} \vdash e_1}{G \vdash \text{fun } x \to e_1}$$

- case: FUN then we have $e = (\text{fun } x \text{ -> } e_1)$ for some x, e_1 and, by induction, we have $fv(e_1) \subseteq G \cup \{x\}$, but then we also have $fv(\text{fun } x \text{ -> } e_1) = fv(e_1) \setminus \{x\} \subseteq ((G \cup \{x\}) \setminus \{x\}) \subseteq G$

$fv(x)$	=	$\{x\}$	
$fv(\text{fun } x \to exp)$	=	$fv(exp) \setminus \{x\}$	*('x' is a bound in exp)*
$fv(exp_1\ exp_2)$	=	$fv(exp_1) \cup fv(exp_2)$	

See tc.ml

STATICALLY RULING OUT PARTIALITY: TYPE CHECKING

Adding Integers to Lambda Calculus

exp ::=
    | …
    | n   *constant integers*
    | $exp_1 + exp_2$ *binary arithmetic operation*

val ::=
    | fun x -> exp   *functions are values*
    | n   *integers are values*

n{v/x} = n   *constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$ *substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \Downarrow (n1\ [\![+]\!]\ n2)}$$

Object-level '+'   Meta-level '+'

NOTE: there are no rules for the case where exp1 or exp2 evaluate to functions!
The semantics is *undefined* in those cases.

Type Checking / Static Analysis

- Recall the interpreter from the Eval3 module:

```
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

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: $x : int, b : bool \vdash if\ (b)\ 3\ else\ x : int$

- 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. $x : int, b : bool \vdash b : bool$
  - 3 must be an int i.e. $x : int, b : bool \vdash 3 : int$
  - x must be an int i.e. $x : int, b : bool \vdash x : int$

Simply-typed Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Recall the free variable calculation
  - Given: G, a map of variable identifiers to types, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e : T$ means "the expression e computes a value of type T, assuming its free variables have the types given in G"

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

"the variable x has type T an is in scope"

$$\frac{G \vdash e_1 : T \to S \qquad G \vdash e_2 : T}{G \vdash e_1\ e_2 : S}$$

"e_1 is a function from T2 to T and e_2 is an expression of type T2"

$$\frac{G, x : T \vdash e : S}{G \vdash \text{fun } (x{:}T) \to e : T \to S}$$

"Given an input of type T, this function computes a result of type S"

Adding Integers

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

INT

$$\frac{}{G \vdash i : \text{int}}$$

VAR

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

ADD

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

FUN

$$\frac{G, x : T \vdash e : S}{G \vdash \text{fun } (x{:}T) \rightarrow e \ : T \rightarrow S}$$

APP

$$\frac{G \vdash e_1 : T \rightarrow S \quad G \vdash e_2 : T}{G \vdash e_1\ e_2 : S}$$

- Note how these rules correspond to the code.
- By convention, if G is empty we leave that spot blank.

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:

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

Example Derivation Tree

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

- 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

$$\vdash (\text{fun } (x{:}\text{int}) \to x\ 3)\ 5\ :\ T$$

$$x : \text{int} \to T \notin x : \text{int}$$

VAR

$$x : \text{int} \vdash x\ :\ \text{int} \to T \qquad x : \text{int} \vdash 3\ :\ \text{int}$$

APP

$$x : \text{int} \vdash x\ 3 : T$$

FUN

$$\vdash (\text{fun } (x{:}\text{int}) \to x\ 3)\ :\ \text{int} \to T \qquad \vdash 5 : \text{int}$$

APP

$$\vdash (\text{fun } (x{:}\text{int}) \to x\ 3)\ 5\ :\ T$$

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