Lecture 14

CIS 341: COMPILERS

Announcements

- Midterm Exam:
  - Not quite done grading yet!
  - Will be available Thursday

- Project 4 is available from the course web pages
  - Due on Thursday, March 21st.
  - As usual, start early and ask questions if you get stuck
  - Note: revised version of LL intermediate representation to be more compliant with “real” LLVM IR

Untyped lambda calculus

Substitution

Evaluation

FIRST-CLASS FUNCTIONS

“Functional” languages

- Languages like ML, Haskell, Scheme, Python, C#, (maybe eventually Java?) include *first-class* functions.
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?

Free Variables and Scoping

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.
- We say that the variable `x` is *free* in `fun y -> x + y`
  - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by "`fun y`" and its scope is the body "`x + y`" in the expression `fun y -> x + y`

- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: λ x. e
- It has variables, functions, and function application.
  - That's it! (Though for examples, I'll add int and +)
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for "functional" languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =
  | Var of var           (* variables             *)
  | Fun of var * exp     (* functions: fun x -> e *)
  | App of exp * exp     (* function application  *)
```

Concrete syntax:

exp ::=	
| x	*variables*
| `fun` x -> exp	*functions*
| exp_1 exp_2	*function application*
| (exp)	*parentheses*

Values and Substitution

- The only values of the lambda calculus are (closed) functions:

```
val ::=
      | fun x -> exp      functions are values
```

- To *substitute* a (closed) value `v` for some variable `x` in an expression `e`
  - Replace all *free occurrences* of `x` in `e` by `v`.
  - In OCaml: written `subst v x e`
  - In Math: written e{v/x}

- Function application is interpreted by substitution:

```
  (fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

Lambda Calculus Operational Semantics

- Substitution function (in Math):

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming y ≠ x)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming y ≠ x)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

x y {(fun z ->z)/y} ⇒ x (fun z -> z)

(fun x -> x y){(fun z -> z) / y} ⇒ (fun x -> x (fun z -> z))

(fun x -> x){(fun z -> z) / x} ⇒ (fun x -> x) // x is not free!

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
    | Var x -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2) -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- A lambda expression `e` is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\text{fun } x \text{ -> } exp) &= fv(exp) \setminus \{x\} \quad \textit{('x' is a bound in exp)} \\ fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2) \end{aligned}$$

Operational Semantics

- Specified using just two inference rules with judgments of the form exp $\Downarrow$ val
  - Read this notation a as "program exp evaluates to value val"
  - This is *call-by-value* semantics: function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \text{ -> } exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \ \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

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 '+'

How to Implement?

- Code in fun.ml shows:
  - A substitution-based interpreter
  - Two environment-based interpreters (one broken)

- We'll come back to compilation after discussing typechecking….

Ruling out ill-defined programs at compile time.

TYPE CHECKING

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 pointer

What to do?

- Don't worry about it… e.g. C, C++
  - Result: segmentation faults, bus errors, etc.

- Make all operations total (i.e. defined everywhere)… e.g. Scheme / Perl
  - 3 + true → 42, … (language specifies behavior)
  - Result: unpredictable answers

- Raise a "runtime type error"… e.g. Python, Ruby, and other dynamically typed languages
  - Result: failure at deployment time

- Try to rule out ill-formed programs… e.g. Java, C#, ML, Haskell
  - 3 + true → compiler error:
    "This expression has type bool but is here used with type int"
  - Result: predictable programs, but it's harder to "get programs running"

- How do you know you've ruled out all ill-formed programs?

Type Soundness

- Build a model of the programming language
  - One model: an interpreter
  - Another model: constructed in mathematics
  - Usually defined via the abstract syntax
- Model defines where the language operations are partial
  - Partiality is different for different languages: e.g. `"foo" + "bar"` is meaningful in Java but not OCaml
- Construct a function: `well_typed : Ast -> unit`
  - When `well_typed e` succeeds, running e will definitely *not* trigger one of the undefined operation cases (i.e. e is type safe)
  - When `well_typed e` aborts with an exception, running e might trigger an undefined operation (i.e. e is not type safe)
- *Prove* that the `well_typed` function is correct.
  - Such proofs are sometimes difficult, but doable for real languages (e.g. SML, Java)

Typechecking

- How do we implement the function `well_typed`?
- Big idea: "approximate" the interpreter:
  - Problem is partiality in the language semantics as defined by the interpreter.
  - Instead of interpreting the program, write a function called `typecheck` that computes a type for the program (rather than the answer obtained by running the program).
  - Behavior of `typecheck` is guided by what the interpreter would do.
- See "tc.ml"

Notes about this Typechecker

- In the interpreter, we only evaluate the body of a function when it's applied.
- In the typechecker, we always check the body of the function (even if it's never applied.)
  - Because of this, we must *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 also determine what type e_1 will return.

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

Defining Type Systems Mathematically

- In the OCaml implementation we have:

```
typecheck (env:environment) (e:exp):ty
```

  - Where `exp` is the type of abstract syntax and `environment` is a `list` of `var * ty` pairs.
  - The result of `typecheck` is a type

- We can abstract this function in math as a relation:
  The notation: $E \vdash e : t$ means typecheck C e = t
  - "In the environment E, program e is well-typed and has type t"
  - "e : t" is a *type judgment*

- Simple examples: $\vdash 3 : \text{int}$ $\quad \vdash \text{true} : \text{bool}$ $\quad \vdash \text{"hello"} : \text{string}$

- Bigger examples: $\vdash (2 * 3) + 5 : \text{int}$ $\quad \vdash \text{if (true) 3 else 4} : \text{int}$

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

Generalizing 'if' & Inference Rules

- For any environment E, expressions e_1, e_2, e_3, and type T the judgment

$$E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T$$

is true if $E \vdash e_1 : \text{bool}$, and $E \vdash e_2 : T$, and $E \vdash e_3 : T$ are all true.

- More succinctly: we summarize this as an *inference rule*:

Premises

$$\frac{E \vdash e_1 : \text{bool} \qquad E \vdash e_2 : T \qquad E \vdash e_3 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

Conclusion

- This rule holds for *any* substitution of the syntactic metavariables E, e_1, e_2, e_3, and T

Simply-typed Lambda Calculus

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

INT

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

VAR

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

ADD

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

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:

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

Example Derivation Tree

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

- 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 $x : \mathrm{int} \in E$ is implemented by the function `lookup`