CIS 341: COMPILERS

Lecture 15
Announcements

• Reminder: HW4 Compiling OAT v.1
• DUE: Thursday, March 26th
• **START TODAY! (IF YOU HAVEN’T ALREADY)**

• Midterm has been graded

• My office hours today: 4:00 – 5:30 instead of 3:30 – 5:00
Average: ~58
Median: ~61
High: 78
(out of 80)
The Story So Far

- As of HW4:
  - See how to compile a C-like language to x86 assembly by way of the LLVM IR

- Main idea 1:
  - Translation by way of a series of languages, each with well-defined semantics

- Main idea 2:
  - Structure of the semantics (e.g. scoping and/or type-checking rules) guides the structure of the translation
What’s next?

• Source language features:
  – First-class functions
  – Objects & Classes
  – Polymorphism
  – Modules

⇒ How do we define their semantics? How do we compile them?

• Performance / Optimization:
  – How can we improve the quality of the generated code?
  – What information do we need to do the optimization?

⇒ Static analyses
Untyped lambda calculus
Substitution
Evaluation
“Functional” languages

- Languages like ML, Haskell, Scheme, Python, C#, Java 8, Swift
- 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

```ml
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?
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: \( \lambda x. e \)
- It has variables, functions, and function application.
  - That’s it!
  - 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
  | exp1 exp2             function application
  | ( exp )               parentheses
```
Values and Substitution

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

\[
\text{val ::=} \\
| \text{fun } x \to \text{exp} \quad \text{functions are values}
\]

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

• Function application is interpreted by \textit{substitution}:

\[
(f \text{un } x \to f \text{un } y \to x + y) \; 1 \\
= \text{subst } 1 \; x \; (f \text{un } y \to x + y) \\
= (f \text{un } y \to 1 + y)
\]
Lambda Calculus Operational Semantics

- Substitution function (in Math):

\[
\begin{align*}
  x\{v/x\} &= v && \text{(replace the free } x \text{ by } v) \\
  y\{v/x\} &= y && \text{(assuming } y \neq x) \\
  (\text{fun } x \rightarrow \text{exp})\{v/x\} &= (\text{fun } x \rightarrow \text{exp}) && \text{(} x \text{ is bound in } \text{exp)} \\
  (\text{fun } y \rightarrow \text{exp})\{v/x\} &= (\text{fun } y \rightarrow \text{exp}\{v/x\}) && \text{(assuming } y \neq x) \\
  (e_1 \; e_2)\{v/x\} &= (e_1\{v/x\} \; e_2\{v/x\}) && \text{(substitute everywhere)}
\end{align*}
\]

- Examples:

\[
\begin{align*}
  x \; y \; ((\text{fun } z \rightarrow z)/y) & \Rightarrow x \; (\text{fun } z \rightarrow z) \\
  (\text{fun } x \rightarrow x \; y)((\text{fun } z \rightarrow z) \; / \; y) & \Rightarrow (\text{fun } x \rightarrow x \; (\text{fun } z \rightarrow z)) \\
  (\text{fun } x \rightarrow x)((\text{fun } z \rightarrow z) \; / \; x) & \Rightarrow (\text{fun } x \rightarrow x) \quad // x \text{ is not free!}
\end{align*}
\]
Free Variable Calculation

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

```ocaml
let rec free_vars (e:exp) : var list =
    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 $\text{free-vars } e$ returns $\text{VarSet.empty}$

- In mathematical notation:

$$
\begin{align*}
\text{fv}(x) &= \{x\} \\
\text{fv}(\text{fun } x \rightarrow \text{exp}) &= \text{fv(exp)} \setminus \{x\} \quad (\text{\textquote{\textquote{x} is a bound in exp}}) \\
\text{fv(exp}_{1}\text{exp}_{2}) &= \text{fv(exp}_{1}\text{)} \cup \text{fv(exp}_{2}\text{)}
\end{align*}
$$
Operational Semantics

• Specified using just two inference rules with judgments of the form $\text{exp} \downarrow \text{val}$
  – Read this notation a as “program exp evaluates to value val”
  – This is call-by-value semantics: function arguments are evaluated before substitution

\[ \text{v} \downarrow \text{v} \]
“Values evaluate to themselves”

\[ \text{exp}_1 \downarrow (\text{fun x -> exp}_3) \quad \text{exp}_2 \downarrow \text{v} \quad \text{exp}_3\{\text{v/x}\} \downarrow \text{w} \]

\[ \text{exp}_1 \text{exp}_2 \downarrow \text{w} \]
“To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function.”
IMPLEMENTING THE
INTERPRETER

See fun.ml
Adding Integers to Lambda Calculus

exp ::= 
| ...  
| n    \textit{constant integers}  
| \text{exp}_1 + \text{exp}_2 \textit{binary arithmetic operation}  

val ::= 
| \text{fun} x \rightarrow \text{exp} \textit{functions are values}  
| n \textit{integers are values}  

\text{n}\{v/x\} = n \textit{constants have no free vars.}  
(\text{e}_1 + \text{e}_2)\{v/x\} = (\text{e}_1\{v/x\} + \text{e}_2\{v/x\}) \textit{substitute everywhere}  

\begin{align*} 
\text{exp}_1 \Downarrow n_1 & \quad \text{exp}_2 \Downarrow n_2 \\
\text{exp}_1 + \text{exp}_2 \Downarrow (n_1 \llbracket + \rrbracket n_2) \\
\end{align*}  

Object-level ‘+’ \quad Meta-level ‘+’