Lecture 5 – Run-time compilation (cont.)

• Compositional compilation
• Adding declarations
• Jumbo, a compositional compiler for Java
Ex: translation to stack machine

- Translation under version 2: Code = \( \text{Var}^* \times \text{ML}_{\text{aug}} \)
  - \( \text{intconst}(n) = ([], \text{PUSHIMM} \, n) \)
  - \( \text{var}(x) = ([x], \text{PUSH} \, x) \)
  - \( \mathcal{+}(\langle x_1, k_1 \rangle, \langle x_2, k_2 \rangle) = \langle x_1 \cup x_2, k_1; k_2; \text{ADD} \rangle \)
  - \( \mathcal{<}(\langle x_1, k_1 \rangle, \langle x_2, k_2 \rangle) = \langle x_1 \cup x_2, k_1; k_2; \text{LT} \rangle \)
  - \( \mathcal{asgn}(x, c) = (\{ x \} \cup x, \langle , k ; \text{POP} \, x \rangle \)
  - \( \mathcal{seq}(c_1, c_2) = \)
  - \( \mathcal{if}(c_1, c_2) = \)

\[
\mathcal{c} = (x, k)
\]

- Finish compilation: \( \text{compile}(S) = \text{substitute} \, \rho(x) \, \text{for each occurrence of} \, x \, \text{in} \, \text{comp}(S) \).

\[
(\_, \langle x, k \rangle)
\]
V. 1 vs. V. 2

- V. 2 potentially more efficient – don’t have to “generate” code at run time – just copy it.
- V. 2 potentially more scope for machine-level optimizations at compile-time (i.e. before generation time).
- However, V. 2 not as general. E.g. not clear how to handle “+” in Java, where different types of arguments can lead to very different code sequences.
Handling declarations

• The environment doesn’t come from thin air. Parts of programs contain declarations that produce the environment for other parts.

• Declarations of names usually come before uses, but not always. E.g. fields can be declared anywhere.

• We will deal only with case where declarations precedes use. Add syntax:

\[
\begin{align*}
    x &\in \text{Ident} \\
    n &\in \text{integers} \\
    e &\in \text{Expr} = x \mid n \mid e + e' \mid e < e' \\
    s &\in \text{Stmt} = x := e \mid s; s' \mid \text{if } (e) \text{ s Stmt} \mid \text{int } x \mid \text{float } x
\end{align*}
\]

• Need to change definition of “Code”, and then change AS operators accordingly.
Handling declarations, v. 1

- Code = Environment $\rightarrow$ ML x Environment

- Define comp:
  - intdecl $= \text{floatdecl} = \lambda \rho. \ ([], \rho[x \rightarrow \text{newloc}(\rho)])$
  - intconst(n) $= \lambda \rho. \ (\text{PUSHIMM} \ n, \rho)$
  - var(x) $= \lambda \rho. \ (\text{PUSH} \ \rho(x), \rho)$
  - +\(c1,c2\) $= \lambda \rho. \ \text{let} \ (M1, _) = c1(\rho), (M2, _) = c2(\rho)$
    \hspace{1cm} $\text{in} \ (M1; M2; \text{ADD}, \rho)$

  - ...  
  - seq(c1,c2) $= \lambda \rho. \ \text{let} \ (M1, \rho') = c1(\rho), (M2, \rho'') = c2(\rho')$
    \hspace{1cm} $\text{in} \ (M1; M2, \rho'')$

- compile(S) $= \text{comp}(S)(\rho_\emptyset)$
Handling declarations, v. 2

- Translation under version 2: Code = Var* × Var* × ML_{aug}
  - intdecl(x) = ([], [x], [])
  - intconst(n) = ([], [], PUSHIMM n)
  - var(x) = ([], [x], PUSH x)
  - +(xs1,k1), (xs2,k2)) = (xs1 ∪ xs2, [[], k1; k2; ADD)
  - asgn(x,c) =
  - seq(c1,c2) = (u1 ∪ u2, d1 ∪ d2, k1; k2)
  - if(c1,c2) =
    
- Finish compilation: compile((U,D,ML)) = let ρ(x) = position of x within D; substitute ρ(x) for each occurrence of x in comp(S).
Compositionality reviewed

• No perfect compositionality—we need a non-compositional function, which we’ve called compile, to “finish off” the compilation. Rather, the idea is to do as much as possible compositionally, because this work can be done at compile time and thus save time at run time.

• A compositional compiler—i.e. definitions of the operators ō—works as well statically as dynamically.
Compositionality reviewed

- Some things are difficult to define compositionally:
  - In Java, the “+” operator cannot do very much, because the code sequences generated by + vary greatly depending upon the types of the arguments, which are generally not known from the arguments alone.
  - Preprocessors make compositionality very difficult, because programs cannot be parsed (until after the preprocessor has run).
  - Global optimizations are difficult because they rely on very detailed knowledge of the context of a statement or expression.
- Java lacks a preprocessor, and Java compilers do little optimization, so it is a good candidate for compositional compilation.
Jumbo

- Jumbo is a compositional compiler for Java (version 2.0, i.e. without generics)
- We will discuss Jumbo in detail in a future class, but we can give (roughly) the definition of Code:

  \[
  \text{Code} = \text{ExportedDefns} \times (\text{Environment} \rightarrow \text{ClosedCode})
  \]

  \[
  \text{ExportedDefns} = \text{ClassInfo} + \text{MethodInfo} + \text{FieldInfo}
  \]

  \[
  \text{Environment} = \text{stack of (ClassInfo} + \text{MethodInfo} + \text{LocalInfo}
  \]

  \[
  \text{ClosedCode} = \text{JVMcode} \times \text{Type} \times \text{Value} \times \text{VarDecls}
  \]