TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

Interpreters

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

Direct Interpretation

- Given the program source code and the run-time input,
- Interpret the source code directly,
  i.e. parse and simulate it, statement by statement
  (syntax-directed interpretation)
  - UNIX shells (command line interpreter)
  - Early interpreters for BASIC, LISP, APL
- Symbol table
  - contains also storage for run-time values of program variables
- Full information about source-level program entities
  - Good for debugging
- Very slow
  - But ok for small scripts

Hybrid Compiler/Interpreter Scenario

Step 1:

- Translate the source program to an internal form
  - E.g. quadruples, postfix, abstract syntax tree
- Or to instructions for an abstract machine
  - E.g. P-code for Pascal and Modula-2, Diana for Ada,
    JVM bytecode for Java, CIL for C#/.NET

Step 2:

- Execute the interpreter
  - given the internal form / abstract machine program
  - simulate the abstract machine step by step

☺ More efficient than direct interpretation, but
☹ still much slower than compiled code, typ. by a factor ~10 to ~100
☺ Still portable – intermediate form is not processor specific
☺ ☹ Source code cannot be reconstructed completely from intermediate form
☺ Can be stored compactly
☺ Easy to write an interpreter (virtual machine)

Example: JVM Bytecode

- Instructions for the **JVM (Java Virtual Machine)**,
  an abstract stack machine
  - Executes .class or .jar files (loaded when first referenced)
    - Heap of loaded classes (program text and static data)
  - Program counter PC
  - Bytecode instructions (postfix order) have
    1 byte opcode with 0 or 1 operand
    - span 1 or more bytes, depending on operand size
  - Run-time stack: Frame pointer fp, Stack pointer sp

☺ Could even be implemented in hardware (e.g. Sun MAJC)

JVM Bytecode Interpretation

JVM Instruction (examples)	Interpretation (by C code)	Stack top before	Stack top afterwards
iconst_0	Stack[sp++] = 0; PC++; // code needs 1 byte	() = don't care	(I) = int-value
istore *v*	Stack[fp + *v*] = Stack[--sp]; PC += 2; // needs 2 bytes	(I)	()
iload *v*	Stack[sp++] = Stack[fp + *v*]; PC += 2;	()	(I)
iadd	Stack[sp-1] = Stack[sp] + Stack[sp-1]; sp--; PC++;	(I, I)	(I)
goto *a*	PC = *a*;	()	()
ifeq *a*	if (Stack[sp--] == 0) PC = *a*; else PC += 3;	(I)	()

Just-In-Time (JIT) Compiling

- A.k.a. **dynamic translation**
- Program execution starts in interpreter as before
- Whenever control flow enters a new **unit** of bytecode
  (unit could be e.g. a class file, a function, a loop, or a basic block):
  - Do not interpret it, but call the JIT compiler that translates it to target code and replaces the unit with a branch to the new target code
- JIT compiling overhead → delay at run-time
  - paid once per unit (if code can be kept in memory)
  - pays often only off if translated code is executed several times
    (e.g., a loop body)
    - Can also be done lazily: Interpret the unit when executed for the first time. When re-entering the unit, JIT-compile.
    - Or pre-compile/pre-JIT to native code ahead of time
  - Trade-off:
    JIT-generated code quality vs. JIT compiler speed (run-time delay)

Just-In-Time (JIT) Compiling (cont.)

- Typically performance boost by at least one order of magnitude
- Typically still somewhat slower,
  but may even be faster than statically compiled code in some cases
  - Can use on-line information from performance counters (e.g. #cache misses) for dynamic re-optimization and memory re-layout
- Example for Java: Sun JDK HotSpot JVM;
  for C#: .NET CLR, NGEN