SWEN430 - Compiler Engineering (2018)

Lecture 7 - Building an Interpreter

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Building an Interpreter

- Writing an interpreter is a good way to prototype an implementation of a f new programming language,
- to help understand their semantics, explore different options,
- and what would be needed in a compiler implementation.
- Many languages like LISP, Prolog and Haskell only existed in interpreted form for several years before adequate compilation techniques were developed.
- Also a good way to implement good debugging tools.

Building an Interpreter — What do we need?

- Representation for programs : AST or some sort of VM code.
  - Pascal P-code led to wide-spread use of Pascal.
  - WAM (Warren Abstract Machine) led to partial compilation for Prolog.
- Associations between names program components, such as methods, method parameters, type definitions.
- Data storage: usually *stack* for local variables, and *heap* for dynamic memory.
  Maybe accessed directly via variable names, or may translate variable references to memory addresses/offsets.
- Control mechanism for recording current location in program, parameters, return values and return addresses for subprograms (functions, subroutines, procedures, methods, ...).

While Interpreter

- Program representation: Use AST produced by parser.
- Interpreter keeps global list of type and method declarations.
- Create a *frame* for each method invocation to store parameter and local variable values: map from names to values.

  The interpreter's run time stack organises these into a stack structure.
- No dynamic storage, so no need for a heap.
- Interpreter's program control manages control within the While program.

While Interpreter

- Interpreter has a method for each form of statement, expression, etc.
- Logic of interpreter methods encodes While language semantics.

  E.g. for `if` statment, evaluate the *condition*; if that is true, execute the *true branch*, otherwise execute the *else branch*.
- Each method returns a result.
  - For expressions: the value of the expression.
  - For statements:
    - Return value for `return` statements.
    - Special value for `break` and `continue` statements.
    - `null`
    - `Collections.EMPTY_SET`

How are these used?

⟹ While interpreter code.

Error Checking

- How can we ensure that the interpreter (or compiled code) never gets a run-time error?

  i.e. an error detect by hardware or operating system sofware — outside of the PL run-time

- Dynamic checking: Check for errors at run time

  i.e. in the interpreter, or compiled code.

- Static checking: Check for errors at compiler time or load time

  i.e. before the program starts executing

- What kinds of errors can/should be detected statically?

Type Checking While Programs Revisited

- We saw several type conditions for While programs:
  - Condition in while and if must be Boolean.
  - Operands of +, - , * and / must be numeric.
  - Operands on `&&` and `||` must be Boolean.
  - In $e_1[e_2]$, e_1 must be an array and e_2 an integer.
  - In $e.f$, e must be a record with field f.
- What others are there?

Type Checking for While

- The expression on the right-hand side of an assignment must have the same type as the variable on the lef-hand side.
- The type of the expression in a `return` statement must be the same as the return type of the enclosing method.
- Each argument in a method call must have the same type as the corresponding parameter in the header of the method being called.

But ...

What does “same type” mean?

Type Checking for While

Consider:

```
type apple is int

...

apple x = 0;

x = x+1;
```

Is this valid?

Type Checking for While

Consider:

```
type apple is int

type orange is int

...

apple x = 0;

orange y = 5

x = x+y;
```

Is this valid?

Are `apple` and `orange` distinct types? Or both the same as `int`?