Chapter 5: Interpreters

Aarne Ranta

Slides for the book "Implementing Programming Languages. An Introduction to Compilers and Interpreters", College Publications, 2012.

The missing piece of a complete language implementation: run your programs and see what they produce.

The code is almost the same as the type checker, thanks to syntax-directed translation.

Not customary for Java or C, but more so for JavaScript.

The quickest way to get your language running.

We will also show an interpreter for the Java Virtual Machine (JVM).

All the concepts and tools needed for solving Assignment 3za.

Specifying an interpreter

Inference rules: **operational semantics**.

The rules tell how to **evaluate** expressions and how to **execute** statements and whole programs.

The judgement form for expressions is

$$\gamma \vdash e \Downarrow v$$

Read: *expression e evaluates to value v in environment* γ.

Values and environments

Values can be seen as a special case of expressions, mostly consisting of literals.

The environment γ (which is a small Γ) assigns values to variables:

$$x_1 := v_1, \ldots, x_n := v_n$$

When evaluating a variable expression, we look up its value from γ:

$$\frac{}{\gamma \vdash x \Downarrow v} \text{ if } x := v \text{ in } \gamma$$

Example: multiplication

Evaluation rule:

$$\frac{\gamma \vdash a \Downarrow u \quad \gamma \vdash b \Downarrow v}{\gamma \vdash a * b \Downarrow u \times v}$$

where $\times$ is multiplication on the level of values.

Typing rule:

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : t}{\Gamma \vdash a * b : t}$$

One view: typing is a special case of evaluation, where the value is always the type!

From rule to code

Pseudocode:

$$
\begin{array}{l}
\mathit{eval}(\gamma, a{*}b) : \\
\quad u := \mathit{eval}(\gamma, a) \\
\quad v := \mathit{eval}(\gamma, b) \\
\quad \textbf{return}\ u \times v
\end{array}
$$

Haskell:

```
eval env (EMul a b) = do
  u <- eval env a
  v <- eval env b
  return (u * v)
```

Java:

```
public Integer visit (EMul p, Env env) {
  Integer u = eval(p.exp_1, env) ;
  Integer v = eval(p.exp_2, env) ;
  return u * v ;
}
```

Side effects

Evaluation can have **side effects** - do things other than return a value.

- change the environment
- perform output

Example: the expression `x = 3` on one hand returns the value 3, on the other changes the value of `x` to 3 in the environment.

The general form of judgement: return a value *and* a new environment γ'.

$$\gamma \vdash e \Downarrow \langle v, \gamma' \rangle$$

Read: *in environment* γ, *expression e evaluates to value v and to new environment* γ'.

Assignment expressions

$$\frac{\gamma \vdash e \Downarrow \langle v, \gamma' \rangle}{\gamma \vdash x = e \Downarrow \langle v, \gamma'(x := v) \rangle}$$

The notation $\gamma(x := v)$ means that we **update** the value of x in γ to v. This **overwrites** any old value of x.

Increments

Operational semantics explains the difference between **pre-increments** (`++x`) and **post-increments** (`x++`).

$$\frac{}{\gamma \vdash \texttt{++}x \Downarrow \langle v+1, \gamma(x := v+1)\rangle} \text{ if } x := v \text{ in } \gamma$$

$$\frac{}{\gamma \vdash x\texttt{++} \Downarrow \langle v, \gamma(x := v+1)\rangle} \text{ if } x := v \text{ in } \gamma$$

Propagation of side effects

Side effects "contaminate" the rules for all expressions.

Example: if we start with $x := 1$, what is the value of

```
x++ - ++x
```

Use the interpretation rule for subtraction:

$$\frac{\gamma \vdash a \Downarrow \langle u, \gamma' \rangle \quad \gamma' \vdash b \Downarrow \langle v, \gamma'' \rangle}{\gamma \vdash a - b \Downarrow \langle u - v, \gamma'' \rangle}$$

Build a proof tree:

$$\frac{x := 1 \vdash \texttt{x++} \Downarrow \langle 1, x := 2\rangle \quad x := 2 \vdash \texttt{++x} \Downarrow \langle 3, x := 3\rangle}{x := 1 \vdash \texttt{x++ - ++x} \Downarrow \langle -2, x := 3\rangle}$$

Statements

Statements are executed for their side effects, not to receive values.

Lists of statements are executed in order

- each statement may change the environment for the next one.

The evaluation of a sequence

$$\gamma \vdash s_1 \ldots s_n \Downarrow \gamma'$$

reduces to the interpretation of single statements by the following two rules:

$$\frac{\gamma \vdash s_1 \Downarrow \gamma' \quad \gamma' \vdash s_2 \ldots s_n \Downarrow \gamma''}{\gamma \vdash s_1 \ldots s_n \Downarrow \gamma''} \qquad \gamma \vdash \ \Downarrow \gamma \quad \text{(empty sequence)}$$

Expression statements

Just ignore the value of the expression:

$$\frac{\gamma \vdash e \Downarrow \langle v, \gamma' \rangle}{\gamma \vdash e; \Downarrow \gamma'}$$

If statements

Difference from the type checker: consider the two possible values of the condition expression.

$$\frac{\gamma \vdash e \Downarrow \langle 1, \gamma' \rangle \quad \gamma' \vdash s \Downarrow \gamma''}{\gamma \vdash \texttt{if}\,(e)\,s\,\texttt{else}\,t \Downarrow \gamma''} \qquad \frac{\gamma \vdash e \Downarrow \langle 0, \gamma' \rangle \quad \gamma' \vdash t \Downarrow \gamma''}{\gamma \vdash \texttt{if}\,(e)\,s\,\texttt{else}\,t \Downarrow \gamma''}$$

Exactly one of them matches at run time. Notice the side effect of the condition!

While statements

If true, then loop. If false, terminate:

$$\frac{\gamma \vdash e \Downarrow \langle 1, \gamma' \rangle \quad \gamma' \vdash s \Downarrow \gamma'' \quad \gamma'' \vdash \texttt{while}\,(e)\,s \Downarrow \gamma'''}{\gamma \vdash \texttt{while}\,(e)\,s \Downarrow \gamma'''} \qquad \frac{\gamma \vdash e \Downarrow \langle 0, \gamma' \rangle}{\gamma \vdash \texttt{while}\,(e)\,s \Downarrow \gamma'}$$

Declarations

Extend the environment with a new variable, which is first given a "null" value.

$$\frac{}{\gamma \vdash t\, x; \Downarrow \gamma, x := \mathit{null}}$$

Using the null value results in a run-time error, but this is of course impossible to completely prevent at compile time (why?).

We don't need to check for the freshness of the new variable, because this has been done in the type checker!

(Cf. Milner: "well-typed programs can't go wrong").

Block statements

Push a new environment on the stack, just as in the type checker.

The new variables declared in the block are popped away at exit from the block.

$$\frac{\gamma. \vdash s_1 \ldots s_n \Downarrow \gamma'.\delta}{\gamma \vdash \{s_1 \ldots s_n\} \Downarrow \gamma'}$$

The new part of the storage,δ, is discarded after the block.

But the old γ part may have changed values of old variables!

Example

```
{
  int x ;          // x := null
  {                // x := null.
    int y ;        // x := null. y := null
    y = 3 ;        // x := null. y := 3
    x = y + y ;    // x := 6. y := 3
  }                // x := 6
  x = x + 1 ;      // x := 7
}
```

Entire programs

C convention: the entire program is executed by running its `main` function, which takes no arguments.

This means the evaluation the expression

```
main()
```

Which means building a proof tree for

$$\gamma \vdash \texttt{main()} \Downarrow \langle v, \gamma' \rangle$$

γ is the **global environment** of the program

- no variables (unless there are global variables)
- all functions: we can look up any function name f and get the parameter list and the function body.

Function calls

Execute the body of the function in an environment where the parameters have the values of the arguments:

$$\frac{\begin{array}{rl} \gamma & \vdash a_1 \Downarrow \langle v_1, \gamma_1 \rangle \\ \gamma_1 & \vdash a_2 \Downarrow \langle v_2, \gamma_2 \rangle \\ \ldots & \\ \gamma_{m-1} & \vdash a_m \Downarrow \langle v_m, \gamma_m \rangle \\ \gamma.x_1 := v_1, \ldots, x_m := v_m & \vdash s_1 \ldots s_n \Downarrow \langle v, \gamma' \rangle \end{array}}{\gamma \vdash f(a_1, \ldots, a_n) \Downarrow \langle v, \gamma_m \rangle}$$

if $t\ f(t_1\, x_1, \ldots, t_m\, x_m)\{s_1\ \ldots, s_n\}$ in γ

This is quite a mouthful. Let us explain it in detail:

- The first m premisses evaluate the arguments of the function call. As the environment can change, we show m versions of γ.
- The last premiss evaluates the body of the function. This is done in a new environment, which binds the parameters of f to its actual arguments.
- No other variables can be accessed when evaluating the body. Hence the local variables in the body won't be confused with the old variables in γ. Actually, the old variables cannot be updated either. All this is already guaranteed by type checking. Thus the old environment γ is needed here only to look up functions, and using γ_m instead of γ here wouldn't make any difference.
- The value that is returned by evaluating the body comes from the `return` statement in the body.

Return values

The value of a function body comes from its `return` statement:

$$\frac{\gamma \vdash s_1 \ldots s_{n-1} \Downarrow \gamma' \quad \gamma' \vdash e \Downarrow \langle v, \gamma'' \rangle}{\gamma \vdash s_1 \ldots s_{n-1}\ \texttt{return}\ e; \Downarrow \langle v, \gamma'' \rangle}$$

If "well-typed programs can't go wrong", the type checker must make sure that function bodies have returns.

Evaluation strategies

Call by value:

- arguments are evaluated *before* the function body is evaluated.
- followed by functions in C

Alternative: **call by name**

- arguments are inserted into the function body as *expressions*, before evaluation
- followed by functions in Haskell (and, in a sense, by macros in C)

Call by name is also known as **lazy evaluation**

- advantage: an argument that is not needed is not evaluated
- disadvantage: if the variable is used more than once, it has to be evaluated again

Call by need avoids the disadvantage, and is actually used in Haskell

Lazy evaluation of boolean expressions

Also in C and Java: `a && b` and `a || b` are evaluated lazily.

In `a && b`, `a` is evaluated first. If the value is false (0), the whole expression comes out false, and `b` is not evaluated at all.

This allows the programmer to write

```
x != 0 && 2/x > 1
```

which would otherwise result in a division-by-zero error when `x == 0`.

Semantic of "and" and "or"

Similar to "if" statements: two rules are needed

$$\frac{\gamma \vdash a \Downarrow \langle 0, \gamma' \rangle}{\gamma \vdash a \&\& b \Downarrow \langle 0, \gamma' \rangle} \qquad \frac{\gamma \vdash a \Downarrow \langle 1, \gamma' \rangle \quad \gamma' \vdash b \Downarrow \langle v, \gamma'' \rangle}{\gamma \vdash a \&\& b \Downarrow \langle v, \gamma'' \rangle}$$

Similarly for `a || b`, where the evaluation stops if `a == 1`.

Implementing the interpreter

Mostly a straightforward variant of the type checker.

Biggest difference: return types and environment:

$\langle \mathit{Val}, \mathit{Env} \rangle$	*eval*	(*Env* γ, *Exp* e)
Env	*exec*	(*Env* γ, *Stm* s)
Void	*exec*	(*Program* p)
Val	*lookup*	(*Ident* x, *Env* γ)
Def	*lookup*	(*Ident* f, *Env* γ)
Env	*extend*	(*Env* γ, *Ident* x, *Val* v)
Env	*extend*	(*Env* γ, *Def* d)
Env	*newBlock*	(*Env* γ)
Env	*exitBlock*	(*Env* γ)
Env	*emptyEnv*	()

The top-level interpreter

First gather the function definitions to the environment, then executes the `main` function:

$$
\begin{array}{l}
\mathit{exec}(d_1 \ldots d_n) : \\
\quad \gamma_0 := \mathit{emptyEnv}() \\
\quad \textbf{for } i = 1, \ldots, n : \\
\qquad \gamma_i := \mathit{extend}(\gamma_{i-1}, d_i) \\
\quad \mathit{eval}(\gamma_n, \texttt{main()})
\end{array}
$$

Statements and expressions

Easier now, because we don't have to decide between type checking and type inference!

Examples:

$exec(\gamma, e;)$:
 $\langle v, \gamma' \rangle := eval(\gamma, e)$
 return γ'

$exec(\gamma, \texttt{while}\ (e)\ s)$:
 $\langle v, \gamma' \rangle := eval(\gamma, e)$
 if $v = 0$
  return γ'
 else

$\gamma'' := \mathit{exec}(\gamma', s)$
$\mathit{exec}(\gamma'', \texttt{while}\ (e)\ s)$

$\mathit{eval}(\gamma, a - b)$:
$\langle u, \gamma' \rangle := \mathit{eval}(\gamma, a)$
$\langle v, \gamma'' \rangle := \mathit{eval}(\gamma', b)$
return $\langle u - v, \gamma'' \rangle$

$\mathit{eval}(\gamma, f(a_1, \ldots, a_m))$:
for $i = 1, \ldots, m$:
$\langle v_i, \gamma_i \rangle := \mathit{eval}(\gamma_{i-1}, a_i)$
$t\ f(t_1\, x_1, \ldots, t_m\, x_m)\{s_1 \ldots s_n\} := \mathit{lookup}(f, \gamma)$
$\langle v, \gamma' \rangle := \mathit{eval}(x_1 := v_1, \ldots, x_m := v_m, s_1 \ldots s_n)$
return$\langle v, \gamma_m \rangle$

Predefined functions

In Assignment 3: input and output of reading and printing integers, doubles, and strings

Implementation: make the `eval` function, call the host language printing or reading functions:

eval(γ, `printInt`(e)) :
 $\langle \gamma', v \rangle$:= *eval*(γ, e)
 // print integer v to standard output
 return $\langle$*void-value*$, \gamma' \rangle$

eval(γ, `readInt`()) :
 // read integer v from standard input
 return $\langle v, \gamma \rangle$

Defining values in BNFC

As an algebraic data type - internal, that is, not parsable:

```
internal VInteger.   Val ::= Integer ;
internal VDouble.    Val ::= Double ;
internal VString.    Val ::= String ;
internal VVoid.      Val ::= ;
internal VUndefined. Val ::= ;
```

You cannot simply write

```
VInteger(2) + VInteger(3)
```

because `+` in Haskell and Java is not defined for the type `Val`.

Instead, a special function `addVal` to the effect that

```
addVal(VInteger(u),VInteger(v)) = VInteger(u+v)
addVal(VDouble(u), VDouble(v))  = VDouble(u+v)
addVal(VString(u), VString(v))  = VString(u+v)
```

In Java, + will do for strings, but in Haskell you need ++.

Implementation in Haskell and Java

Follow the same structure as in Chapter 4.

In Haskell, the IO monad is now the most natural choice.

```
execStm :: Env -> Stm -> IO Env
evalExp :: Env -> Exp -> IO (Val,Env)
```

In Java, the corresponding types are

```
class StmExecuter implements Stm.Visitor<Object,Env> {
  public Object visit(CPP.Absyn.SDecl p, Env env)
  ...
class ExpEvaluator implements Stm.Visitor<Val,Env> {
  public Val visit(CPP.Absyn.EAdd p, Env env)
  ...
```

The Visitor interface expectedly has the return type `Val` in `ExpEvaluator`, and the dummy type `Object` in `StmExecuter`.

The environment can be changed as a side effect in Java. In Haskell, this would also be possible with a **state monad**.

Debugger

An easy variant of the interpreter.

Print the environment (i.e. the values of variables) after each change of the values.

Also print the statement or expression causing the change of the environment.

Interpreting Java bytecode*

Java is *not* an interpreted language!

Java is *compiled* to JVM (= Java Virtual Machine =Java bytecode).

JVM is interpreted.

The interpreter is much simpler than the one described above.

Executing a JVM program

Compiled from the expression 5 + (6 * 7); **stack** on the right of ";"

```
bipush 5  ; 5
bipush 6  ; 5 6
bipush 7  ; 5 6 7
imul      ; 5 42
iadd      ; 47
```

JVM instructions

Machine languages have **instructions**, instead of expressions and statements. Here are some:

instruction	**explanation**
`bipush` *n*	push byte constant *n*
`iadd`	pop topmost two values and push their sum
`imul`	pop topmost two values and push their product
`iload` *i*	push value stored in address *i*
`istore` *i*	pop topmost value and store it in address *i*
`goto` *L*	go to code position *L*
`ifeq` *L*	pop top value; if it is 0 go to position *L*

The instructions working on integers have variants for other types in the full JVM.

Variables and addresses

The code generator assigns an integer **memory address** to every variable.

Declarations are compiled so that the next available address is reserved to the variable declared.

Using a variable as an expression means **loading** it, whereas assigning to it means **storing** it.

Example Java code and JVM:

```
int i ;              ; reserve address 0 for i, no code generated
i = 9 ;              bipush 9
                     istore 0
int j = i + 3 ;      ; reserve address 1 for j
                     iload 0
                     bipush 3
                     iadd
                     istore 1
```

Control structures

E.g. `while` and `if`: compiled with **jump instructions**:

- `goto`, **unconditional jump**,
- `ifeq`, **conditional jump**

Jumps go to **labels**, which are positions in the code.

Example: `while` statements

```
while (exp)
  stm
```

```
TEST:
  ; here, code to evaluate exp
  ifeq END
  ; here, code to execute stm
  goto TEST
END:
```

Defining a JVM interpreter

Transitions, a.k.a. **small-step semantics**: each rule specifies one step of computation.

The operational semantics for C/Java source code above was **big-step semantics**: we said that `a + b` is evaluated so that `a` and `b` are evaluated first; but each of them can take any number of intermediate steps.

The format of a small-step rule for JVM:

$$\langle \mathit{Instruction}, \mathit{Env} \rangle \longrightarrow \langle \mathit{Env'} \rangle$$

Environment *Env*:

- **code pointer** P,
- **stack** S,
- **variable storage** V.

The rules work on instructions, executed one at a time.

The next instruction is determined by the code pointer.

Each instruction can do some of the following:

- increment the code pointer: $P + 1$
- change the code pointer according to a label: $P(L)$
- copy a value from a storage address: $V(i)$
- write a value in a storage address: $V(i := v)$
- push values on the stack: $S.v$
- pop values from the stack

Semantic rules for some instructions

$$
\begin{array}{llcll}
\langle \texttt{bipush}\, v, & P, V, S\rangle & \longrightarrow & \langle P+1, & V, & S.v\rangle \\
\langle \texttt{iadd}, & P, V, S.v.w\rangle & \longrightarrow & \langle P+1, & V, & S.v+w\rangle \\
\langle \texttt{imul}, & P, V, S.v.w\rangle & \longrightarrow & \langle P+1, & V, & S.v\times w\rangle \\
\langle \texttt{iload}\, i, & P, V, S\rangle & \longrightarrow & \langle P+1, & V, & S.V(i)\rangle \\
\langle \texttt{istore}\, i, & P, V, S.v\rangle & \longrightarrow & \langle P+1, & V(i:=v), & S\rangle \\
\langle \texttt{goto}\, L, & P, V, S\rangle & \longrightarrow & \langle P(L), & V, & S\rangle \\
\langle \texttt{ifeq}\, L, & P, V, S.0\rangle & \longrightarrow & \langle P(L), & V, & S\rangle \\
\langle \texttt{ifeq}\, L, & P, V, S.v\rangle & \longrightarrow & \langle P+1, & V, & S\rangle\ (v\neq 0)
\end{array}
$$

Relating the two kinds of semantics

The big-step relation $\Downarrow$ can be seen as the **transitive closure** of the small-step relation $\longrightarrow$:

> $e \Downarrow v$ means that $e \longrightarrow \ldots \longrightarrow v$ in some number of steps.

In the JVM case $e \Downarrow v$ means that executing the instructions in e returns the value v on top of the stack after some number of steps and then terminates.

This makes it possible, in principle, to prove the **correctness of a compiler**:

> An expression compiler c is *correct* if, for all expressions e, $e \Downarrow v$ if and only if $c(e) \Downarrow v$.

Objects and memory management*

Values have different sizes: integers might be 32 bits, doubles 64 bits.

But what is the size of a string?

It has *no* fixed size. If you declare a string variable, the size of its value can grow beyond any limits when the program is running.

This is generally the case with **objects**, which in Java have **classes** as types.

Example: a function that replicates a string k times:

```
string replicate(int k, string s) {
  string r ;
  r = s ;
  int i = 1 ;
  while (i < k){
    r = s + r ;
    i++ ;
  }
  return r ;
}
```

When the string variable `r` is declared, one memory word is allocated to store an **address**.

Loading `r` on the stack means loading just this address - which has the same size as an integer, independently of the string stored.

The address indicates the place where the string is stored. It is not on the stack, but in another part of the memory, called the **heap**.

Run this program with $k = 2$, $s =$ "*hi*".

source	JVM	stack	V	heap
	; k in 0, s in 1	-	2,&s	&s>"hi"
`string r ;`	; reserve 2 for r	-	2,&s,_	
`r = s ;`	`aload 1`	&s		
	`astore 2`	-	2,&s,&s	
`int i = 1 ;`	; reserve 3 for i	-	2,&s,&s,_	
	`bipush 1`	1		
	`istore 3`	-	2,&s,&s,1	
`r = s + r ;`	`aload 1`	&s		
	`aload 2`	&s.&s		
	; call string-+	&r		&s>"hi",&r>"hihi"
	`astore 2`	-	2,&s,&r,1	&s>"hi",&r>"hihi"

The variables in V are **stack variables**, and store values of fixed sizes.

But `s` and `r` are **heap variables**. For them, V stores just addresses to the heap. They can be sharedor or split. Any amount of storage reserved for them may get insufficient.

Memory management

When a function terminates (usually by `return`), its stack storage is freed. But this does not automatically free the heap storage.

For instance, at the exit from the function `replicate`, the storage for `k`, `s`, `r`, and `i` is emptied, and therefore the addresses `&s` and `&r` disappear from V.

But we cannot just take away the strings `"hi"` and `"hihi"` from the heap, because they or their parts may still be accessed by some other stack variables from outer calls.

While a program is running, its heap storage can grow beyond all limits, even beyond the physical limits of the computer. To prevent this, **memory management** is needed.

Manual memory management (C and C++): `malloc` reserves a part of memory, `free` makes it usable again.

Standard Template Library (C++) tries to hide much of this from the application programmer.

Garbage collection (Java, Haskell): automatic. A procedure that finds out what parts of the heap are still needed (i.e. pointed to from the stack).

Mark-sweep garbage collection

Perhaps the simplest of the algorithms.

The stack is an array of two kinds of elements:

- data (an integer value)
- heap address (can also be an integer, but distinguishable from data)

The heap is segmented to **blocks**. Each element in the heap is one of:

- beginning of a block, indicating its length (integer) and freeness (boolean)
- data (an integer value)
- heap address (pointing to elsewhere in the heap)
- unused

Example: an unaddressed block containing an array of the three integers 6, 7, 8 is a heap segment of four elements:

```
begin 3 false, data 6, data 7, data 8
```

The algorithm composes three functions:

- *roots*: find all heap addresses on the stack
- *mark*: recursively mark as `true` all heap blocks that are addressed
- *sweep*: replace unmarked memory blocks with "unused" cells and unmark marked blocks

The boolean freeness value of beginning elements indicates marking in the mark-sweep garbage collection. The mark of the beginning node applies to the whole block, so that the sweep phase either preserves or frees all nodes of the block.