Compilers, Interpreters, and Partial Evaluators

Jens Palsberg

Nov 24, 2016

1 The functionality of a compiler

An L-program comp is an N-to-L-compiler iff

$$\forall p \in P_N : \quad \forall d \in D : \qquad [\![[\![comp]\!]_L (p)]\!]_L (d) = [\![p]\!]_N (d)$$

2 The functionality of an interpreter

An L-program int is an N/L-interpreter iff

$$\forall p \in P_N : \quad \forall d \in D : \qquad [\![int]\!]_L (p, d) = [\![p]\!]_N (d)$$

3 A small language

$$\begin{array}{rcl} e & ::= & input \quad | \quad true \quad | \quad false \quad | \quad e \mathbin{?} e : e \\ v \in D & = & \{true, false\} \end{array}$$

Syntactic sugar:

$$\begin{array}{rcl} e_1 \ \texttt{\&\&}\ e_2 & = & e_1 \mathbin{?} false : e_2 \\ e_1 \ \texttt{||}\ e_2 & = & e_1 \mathbin{?} true : e_2 \\ !\, e & = & e \mathbin{?} false : true \end{array}$$

4 A compiler for a small language

Assume that the input is stored in a variable *input*.

Format of each judgment: $k \vdash e \rightarrow c, k'$.

$$k \vdash input \rightarrow (v_k = input), k+1$$

$$k \vdash true \rightarrow (v_k = true), k+1$$

$$k \vdash false \rightarrow (v_k = false), k+1$$

$$\frac{k+2 \vdash e_1 \to c_1, k_1 \qquad k_1 \vdash e_2 \to c_2, k_2 \qquad k_2 \vdash e_3 \to c_3, k_3}{k \vdash e_1 \ ?\ e_2 : e_3 \to \begin{array}{rl} & c_1 \\ & if\ v_{k+2}\ goto\ else_{k+1} \\ & c_2 \\ & v_k \ = \ v_{k_1} \\ & goto\ end_{k+1} \\ else_{k+1}: & c_3 \\ & v_k \ = \ v_{k_2} \\ end_{k+1}: & \text{// nothing here} \end{array}, k_3}$$

5 A small-step semantics for a small language

$$\frac{e_1 \longrightarrow e_1'}{e_1 \ ?\ e_2 : e_3 \longrightarrow e_1' \ ?\ e_2 : e_3}$$

$$true \ ?\ e_2 : e_3 \longrightarrow e_2$$

$$true \ ?\ e_2 : e_3 \longrightarrow e_3$$

6 A big-step semantics for a small language

$$\frac{e_1 \longrightarrow true \qquad e_2 \longrightarrow v}{e_1 \ ?\ e_2 : e_3 \longrightarrow v}$$

$$\frac{e_1 \longrightarrow false \qquad e_3 \longrightarrow v}{e_1 \ ?\ e_2 : e_3 \longrightarrow v}$$

7 An interpreter for a small language

```
function int(Exp e, boolean b) {
  case e {
    input: return b
    true : return true
    false: return false
    (e1 ? e2 : e3) :
                            v = int(e1,b)
                            if0 v goto else
                            v = int(e2,b)
                            goto end
                     else:  v = int(e3,b)
                      end:  return v
  }
})
```

8 Comparison of the compiler and the interpreter

Compared to the interpreter, the compiler has:

- unrolled the recursion and
- replaced the case expression with the code for each entry.

9 Compiler generators

Can we automatically map the interpreter to the compiler? In other words, the challenge is to write a program cogen such that:

$$cogen(int) \quad = \quad comp$$

Idea:

```
static Function
cogen =
  int -> { p -> { v -> int(p,v) } }
}
```

We want `cogen` to produce something similar to the compiler above.

10 Partial evaluators

An L-program s is an L-specializer iff

$$\forall p \in P_L : \quad \forall x, y \in D : \qquad [\![[\![mix]\!]_L (p, x)]\!]_L (y) = [\![p]\!]_L (x, y)$$

Example:

```
int pow(n, x) {
  int res = 1;
  while (n > 0) { res = res * x; n = n - 1; }
  return res;
}
```

Suppose we know n = 3.

$[\![mix]\!]_L (pow, 3)$

```
 = int pow3(x) {
     int res = 1;
     res = res * x;
     res = res * x;
     res = res * x;
     return res;
   }
```

11 Futamura projections

We label the three Futamura projections by 1^{st}, 2^{nd}, and 3^{rd}. The theorem $[\![cogen]\!]_L(mix) = cogen$ below is known as the 4^{th} Futamura projection.

Definitions:

$$
\begin{array}{lcrcl}
 & & [\![p]\!]_N(x) & = & out \\
 & & [\![int]\!]_L(p, \quad x) & = & out \\
1^{st} & : & [\![mix]\!]_L(int, \quad p) & = & code \\
2^{nd} & : & [\![mix]\!]_L(mix, \quad int) & = & comp \\
3^{rd} & : & [\![mix]\!]_L(mix, \quad mix) & = & cogen
\end{array}
$$

Theorems:

$$
\begin{array}{lcl}
[\![code]\!]_L(x) & = & out \\
[\![comp]\!]_L(p) & = & code \\
[\![cogen]\!]_L(int) & = & comp \\
[\![cogen]\!]_L(mix) & = & cogen
\end{array}
$$

Proofs:

$$
\begin{array}{rl}
 & [\![code]\!]_L(x) \\
= & [\![[\![mix]\!]_L(int, p)]\!]_L(x) \\
= & [\![int]\!]_L(p, x) \\
= & out
\end{array}
$$

$$
\begin{array}{rl}
 & [\![comp]\!]_L(p) \\
= & [\![[\![mix]\!]_L(mix, int)]\!]_L(p) \\
= & [\![mix]\!]_L(int, p) \\
= & code
\end{array}
$$

$$
\begin{array}{rl}
 & [\![cogen]\!]_L(int) \\
= & [\![[\![mix]\!]_L(mix, mix)]\!]_L(int) \\
= & [\![mix]\!]_L(mix, int) \\
= & comp
\end{array}
$$

$$
\begin{array}{rl}
 & [\![cogen]\!]_L(mix) \\
= & [\![[\![mix]\!]_L(mix, mix)]\!]_L(mix) \\
= & [\![mix]\!]_L(mix, mix) \\
= & cogen
\end{array}
$$

End of Proofs.