Model Interpretation and Compilation by Partial Evaluation

William R. Cook
University of Texas at Austin
with
Ben Wiedermann, Ali Ibrahim, Ben Deleware, Sidney Rosairo, Amin Shali, David Kitchin
Model Transformation

Model → Transform → Code
Model Transformation
Do we need generate code?

Model -> Transform -> Code -> Run -> Behavior

Interpret
Interpreters are Slower

Model ➔ Transform ➔ Code ➔ Run ➔ Behavior

Interpret
Get the code from the interpreter?
Yes!!
Partial Evaluation of an interpreter creates compiled code

Model → Transform → Code → Run → Behavior

Interpret → PE
Yes!!
Partial Evaluation of an interpreter creates compiled code

Model → Transform → Code → Run → Behavior

Interpret

Its a long story...
Partial Evaluation (by hand)

Example: Power function

\[ \text{pow}(n, x) = \text{if } (n==0) \text{ then } 1 \text{ else } x \times \text{pow}(n-1, x) \]

What if you know \( n \)?

\[ \text{pow}(3, x) = x \times \text{pow}(2, x) \]

This depends on \( \text{pow}(2, x) \)

\[ \text{pow}(2, x) = x \times \text{pow}(1, x) \]
\[ \text{pow}(1, x) = x \times \text{pow}(0, x) \]
\[ \text{pow}(0, x) = 1 \]
Partial Evaluation

Example: Power function

\[
pow(n, x) = \text{if } (n==0) \text{ then } 1 \text{ else } x \times pow(n-1, x)\]

Let's call this final function “pow3”:

\[
pow3(x) = x \times x \times x\]

Partial evaluation

Eliminates computations that depend on known inputs
Result is “residual program”

Doesn’t always work:

\[
pow(n, 19) = \text{if } (n==0) \text{ then } 1 \text{ else } 19 \times pow(n-1, 19)\]

Useful when raising many numbers to 3rd power
Automatic Partial Evaluation

Example: Power function

\[ \text{pow}(n, x) = \text{if } (n==0) \text{ then } 1 \text{ else } x \cdot \text{pow}(n-1, x) \]

Can we compute residual code automatically?

\[ \text{peval}(\text{pow}, 3) \rightarrow \text{fun}(x) x \cdot x \cdot x \equiv \text{pow}3 \]

Partial evaluation function: peval

Inputs:
- Source code of a function
- Value of the first argument

Output:
- Residual code from partially evaluating
Automatic Partial Evaluation

More formally, for any function $f$ and value $v$

$$\text{peval}( f, v ) = g$$

- “peval” is traditionally called “mix”

such that for any value $x$

$$g(x) = f(v, x)$$

One implementation is “currying”

$$\text{peval}( f, v ) = \lambda x. f(v, x)$$

a true partial evaluator returns residual code, not a curried function
Interpreters

Command line for python interpreter

```python
python notify.pl in.txt > out.txt
```

An interpreter is a function of two arguments

```python
python("notify.pl", "in.txt") ➔ output
```

Just like the pow function

```python
pow(3, 19) ➔ 6,895
```
Partial Evaluation of Interpreters

What if the program is known but input is not?

```python
python("notify.pl", ?)
```

Useful because we often run the same python program many times on different inputs

Apply automatic partial evaluation

```python
peval( python, "notify.pl" ) → g
```

where

```python
g("in.txt") = python("notify.pl", "in.txt")
```

What is "g"?
Partial Evaluation of Interpreters

What if the program is known but input is not?

```
python("notify.pl", ?)
```

Useful because we often run the same python program many times on different inputs

Apply automatic partial evaluation

```
peval( python, "notify.pl" ) \rightarrow g
```

where

```
g("in.txt") = python("notify.pl", "in.txt")
```

What is “g”?

Compiled version of “notify.pl”!
Example Modeling Language/Model

State Machine Model

```java
class State {
    String label;
    Transition[] trans;
}

class Transition {
    String event;
    State to;
}
```

Diagram:
- Open
- Closed
- Open to Closed: open arrow
- Closed to Open: close arrow
Example Model Interpreter

Interpreter

```java
int run(State current) {
    print(current.label);
    String input = in.readLine();
    for (Trans t : current.trans)
        if (t.event == input)
            return run(t.to);
    return run(current);
}
```
Partial Evaluation

int runOpen() {
    print("Open");
    String input = in.readLine();
    if ("close" == input)
        return runClosed();
    return runOpen();
}

int runClosed() {
    print("Closed");
    String input = in.readLine();
    if ("open" == input)
        return runOpen();
    return runClosed();
}

int run(State current) {
    print(current.label);
    String input = in.readLine();
    for (Trans t : current.trans)
        if (t.event == input)
            return run(t.to);
    return run(current);
}
First Futamura Projection (1971)

Partial evaluation of an interpreter with respect to a program is a compiled version of the program.
That is, result is a version of the interpreter specialized to run just that one program.
Futamura Projections I

Interpreter

\[
\text{python(“notify.pl”, “in.txt”) } \rightarrow \ o
\]

First Futamura projection

\[
\text{peval(python, “notify.pl”) } \rightarrow \ g \quad \text{where } g(“in.txt”) = o
\]
Futamura Projections (pattern)

Interpreter

\[
\text{python(“notify.pl”, “in.txt”) } \rightarrow o
\]

First Futamura projection

\[
\text{peval(pyhton, “notify.pl”)} \rightarrow g \quad \text{where } g(“in.txt”) = o
\]

\(g\) is compiled version of notify.pl
Futamura Projections II

Interpreter
python("notify.pl", "in.txt") ➔ o

First Futamura projection
peval(python, "notify.pl") ➔ g where g("in.txt") = o

g is compiled version of notify.pl

Second Futamura projection
peval(peval, python) ➔ c where c("notify.pl") = g
Futamura Projections II

Interpreter

depython("notify.pl", "in.txt") → o

First Futamura projection

depeval(depython, "notify.pl") → g where g("in.txt") = o

g is compiled version of notify.pl

Second Futamura projection

depeval(depython, python) → c where c("notify.pl") = g

c is a python compiler
Futamura Projections III

Interpreter

\[
\text{python}("\text{notify.pl}", "\text{in.txt}") \rightarrow o
\]

First Futamura projection

\[
\text{peval} (\text{python}, "\text{notify.pl}") \rightarrow g \quad \text{where } g("\text{in.txt}") = o
\]

g is compiled version of notify.pl

Second Futamura projection

\[
\text{peval} (\text{peval}, \text{python}) \rightarrow c \quad \text{where } c("\text{notify.pl}") = g
\]

c is a python compiler

Third Futamura projection

\[
\text{peval} (\text{peval}, \text{peval}) \rightarrow z \quad \text{where } z(\text{python}) = c
\]
Futamura Projections III

Interpreter

\[ \text{python(“notify.pl”, “in.txt”) } \Rightarrow \text{ o} \]

First Futamura projection

\[ \text{peval(python, “notify.pl”) } \Rightarrow \text{ g} \quad \text{where g(“in.txt”) = o} \]

\(\text{g} \) is compiled version of notify.pl

Second Futamura projection

\[ \text{peval(peval, python)} \Rightarrow \text{ c} \quad \text{where c(“notify.pl”) = g} \]

\(\text{c} \) is a python compiler

Third Futamura projection

\[ \text{peval(peval, peval)} \Rightarrow \text{ z} \quad \text{where z(python) = c} \]

\(\text{z} \) is a compiler compiler!
We only need First Projection

Interpreter

\[
\text{python(“notify.pl”, “in.txt”)} \rightarrow o
\]

First Futamura projection

\[
\text{peval(python, “notify.pl”)} \rightarrow g \quad \text{where } g(“in.txt”) = o
\]

\[g \text{ is compiled version of notify.pl}\]

Second Futamura projection

\[
\text{peval(peval, python)} \rightarrow c \quad \text{where } c(“notify.pl”) = g
\]

\[c \text{ is a python compiler}\]

Third Futamura projection

\[
\text{peval(peval, peval)} \rightarrow z \quad \text{where } z(\text{python}) = c
\]

\[z \text{ is a compiler compiler!}\]
Avoid Need for Self-Applicable `peval`

**Interpreter**
\[
\text{python(“notify.pl”, “in.txt”) } \rightarrow \text{ o}
\]

First Futamura projection
\[
\text{peval(python, “notify.pl”) } \rightarrow \text{ g} \quad \text{where g(“in.txt”) = o}
\]
g is compiled version of notify.pl

Second Futamura projection
\[
\text{peval(peval, python)} \rightarrow \text{ c} \quad \text{where c(“notify.pl”) = g}
\]
c is a python compiler

Third Futamura projection
\[
\text{peval(peval, peval)} \rightarrow \text{ z} \quad \text{where z(python) = c}
\]
z is a compiler compiler!
Futamura in Practice

Interpreters have “good” behavior
Control flow depends on program first, then input
just like pow(n, x): control flow depends on n

Can’t make good compilers via 2nd/3rd Futamura
Trying to make a C compiler via Futamura will fail
Was that the right goal?
Be careful what you pick as challenge problem

Hypothesis:
First Futamura projection will work well enough for model interpreters
solves real problem, simple partial evaluator
Plan

Goal
Compile model languages
by partial evaluation of model interpreters

Technique
Model interpreters written in Java
  e.g. ModelTalk, WebDSL, others?
Partial Evaluator for Java
Residual (compiled code) is also in Java

So...
How do we write a partial evaluator for Java?
Language Levels

Two levels of language

M: Language being interpreted
  e.g. Python or a Modeling language
L: Language in which interpreter is written
  e.g. Java, C, etc

Components

Partial evaluator for L
Model interpreter I for M written in L
A model R written in M
Residual code peval(I, R) is also in L
Simple Evaluator $E$

$x : \text{Variable} \quad \nu : \text{Value}$

$e = x \mid \nu \mid \text{if } e \text{ then } e \text{ else } e \mid e + e \mid f(e, \ldots, e)$

$\rho$ environment maps \textit{all} variables to values

$E[\nu]\rho = \nu$

$E[x]\rho = \rho(x)$

$E[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\rho =$ $\text{if } E[e_1]\rho \text{ then } E[e_2]\rho \text{ else } E[e_3]\rho$

$E[e_1 + e_2]\rho = E[e_1]\rho + E[e_2]\rho$

$E[f(e_1, \ldots, e_n)]\rho = E[e]\rho'$

\textbf{lookup function definition:} $f(x_1, \ldots, x_n) = e$

$\rho' = \{ x_1 = E[e_1]\rho, \ldots, x_n = E[e_n]\rho \}$
From Full Evaluation to Partial Evaluation

The type of eval

\[ E : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Value} \]

\[ \text{Environment} = \text{Variable} \rightarrow \text{Value} \]
\[ \text{FreeVars}(e) \subseteq \text{Domain}(v) \]

All variables are bound

What about a partial evaluator?

\[ P : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Expression} \]

Result might not be a complete value

\[ P[x+y] \{x=3, y=2\} \rightarrow 5 \]
\[ P[x+y] \{x=3\} \rightarrow [3+y] \]
Online Partial Evaluator $P$

$x$: Variable  \hspace{1cm} $\nu$: Value

$e = x \mid \nu \mid \text{if } e \text{ then } e \text{ else } e \mid e + e \mid f(e, \ldots, e)$

$\rho$ environment maps some variables to values

$P[\nu]\rho = \nu$

$P[x]\rho = \text{if } x \in \text{dom}(\rho) \text{ then } \rho(x) \text{ else } [x]$

returns code $[x]$ if the variable is not defined
Online Partial Evaluator $P$

$x$ : Variable    $v$ : Value

$e = x \mid v \mid \text{if } e \text{ then } e \text{ else } e \mid e+e \mid f(e, \ldots, e)$

$\rho$ environment maps some variables to values

$P[v]\rho = v$

$P[x]\rho = \text{if } x \in \text{dom}(\rho) \text{ then } \rho(x) \text{ else } [x]$

$P[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\rho =$

\[
\text{case } P[e_1]\rho \text{ of }
\]

$\nu \rightarrow \text{if } \nu \text{ then } P[e_2]\rho \text{ else } P[e_3]\rho$

$[e] \rightarrow [\text{if } e \text{ then } P[e_2]\rho \text{ else } P[e_3]\rho]$

if its a boolean $\nu$, then pick branch.
else create a new if statement
Online Partial Evaluator $P$

$P[v]_\rho = v$

$P[x]_\rho = \text{if } x \in \text{dom}(\rho) \text{ then } \rho(x) \text{ else } [x]$

$P[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]_\rho =$

\[
\text{case } P[e_1]_\rho \text{ of }
\]

\[
\nu \rightarrow \text{if } \nu \text{ then } P[e_2]_\rho \text{ else } P[e_3]_\rho
\]

\[
[e] \rightarrow [\text{if } e \text{ then } P[e_2]_\rho \text{ else } P[e_3]_\rho]
\]

$P[e_1 + e_2]_\rho =$

\[
v_1 + v_2 \quad \text{if } v_i = P[e_i]_\rho
\]

\[
[e'_1 + e'_2] \quad \text{if } e'_i = P[e_i]_\rho
\]

apply operator if arguments are both are values

otherwise generate new expression
Function Calls

$P[\mathcal{f} (e_1, \ldots, e_n)]\rho = [\mathcal{f}_{v_1,\ldots,v_j} (e'_{d_1}, \ldots, e'_{d_k})]$

1. lookup function definition: $\mathcal{f} (x_1, \ldots, x_n) = e$
2. Partially evaluate the arguments
   
   $e'_i = P[e_i]$ 
3. partition arguments into static and dynamic
   
   $\{ s_1, \ldots, s_j \} \cup \{ d_1, \ldots, d_k \} = \{ 1, \ldots, n \}$
   
   $\forall \{ e'_{s_1}, \ldots, e'_{s_j} \} = \{ v_1, \ldots, v_j \}$
4. create environment with static variables
   
   $\rho' = \{ x_{s_1} = v_1, \ldots, x_{s_j} = v_j \}$
5. create new function specialized by statics
   
   $\mathcal{f}_{v_1,\ldots,v_j} (x_{d_1}, \ldots, x_{d_k}) = E[e]\rho'$
6. Residual code is call with dynamic arguments
   
   $[\mathcal{f}_{v_1,\ldots,v_j} (e'_{d_1}, \ldots, e'_{d_k})]$
Java Partial Evaluation Concerns

Mutable state

Supported!

A mutable object is either static or dynamic stage

All mutations happen in correct order with stage
(But static stage happens before dynamic stage)

Reflection becomes static

String name = "getSize";
Method m = o.getClass().getMethod(name);
m.invoke(o);

converts to:
o.getSize();