CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 35: Partial Evaluation

18 April 08

Programs and the Functions they Implement

- Let f be a program written in language L
- $[\![\,]\!]_L$ is the program meaning function for language L
- $[\![f]\!]_L$ [in] is the result of executing program f on input in

  output = $[\![f]\!]_L$ [in]

  I.e., output = function $[\![f]\!]_L$ appled to argument in
- Omit subscript L when the language in which f is implemented is unimportant

Interpreters

- An interpreter for language S is a program **int** (written in language L) that has two inputs
  - in_1, a program written in source language S
  - in_2, input to the program in_1

- The result of running **int** on inputs [in_1, in_2] is the output of program in_1 when run on input in_2, i.e.,

$$[\![\mathbf{int}]\!]_L\ [in_1, in_2] = [\![in_1]\!]_S\ [in_2]$$

Partial Evaluators

- An partial evaluator for language S is a program **mix** (written in language L) that has two inputs
  - f, a program (written in source language S) that takes two inputs
  - in_1, the first input for program f

- The result of running **mix** on inputs [in_1, in_2] is a specialized version of program fin_1 that has its first input, in_2, baked in. The program returned by **mix**, when presented with second input in_2, returns $[\![f]\!]_S$ [in_1, in_2]

$$[\![\, [\![\mathbf{mix}]\!]_L\ [f, in_1]\,]\!]_S\ [in_2] = [\![f]\!]_S\ [in_1, in_2]$$

Trivial Example

- Let f be (lambda (x y)(+ x y))
- Let f2 = ⟦**mix**⟧ [f,2]
- f2 could be

  (lambda(d) (f 2 d))

  which is

  (lambda(d) ((lambda (x y)(+ x y)) 2 d))

  but this doesn't count, as partial evaluation is done to speed up computation, not slow it down
- A better f2 would be (lambda (y)(+ 2 y)) which requires propagating the constant 2 into the body of f and making a cloned copy of f that has 2 truly baked in.

A Less Trivial Example

- Let f be

  f(n, x) = **if** n=0 **then** 1 **else**

  if even(n) **then** f(n/2, x) ↑ 2

  else x * f(n-1,x)

- Let f5 = ⟦**mix**⟧ [f,5]
- A good f5 obtained by evaluating the loop at partial-evaluation time would be

  f5(x) = x * ((x ↑2) ↑2)

- Note: partial evaluating w.r.t argument x rather than n results in very little improvement.

A rose by any other name

- Called partial evaluation because the program is only partially executed (with respect to a subset of its inputs) and the rest remains symbolic.
- Historically called “mixed computation”, hence **mix**.
- Essentially, aggressive constant propagation, constant folding, and dead code elimination.
- Essentially, repeated beta-substitution, where the residual program (after no further beta-substitutions can be done) is the result.
- Essentially, elimination of interpretive overhead.
- Related to template meta-programming and template instantiation in C++.

Binding Time Analysis

- Code annotation to distinguish between static and dynamic parts

  f(n, x) = **if** n=0 **then** 1 **else**

  if even(n) **then** f(n/2, x) ↑ 2

  else x * f(n-1,x)

- Non-underlining indicates *static* part that can be evaluated at partial-evaluation time and eliminated
- Underlining indicates *dynamic* content that must survive in the specialized program

⟦**mix**⟧ [f,5]

f(n, $\underline{x}$) = **if** n=0 **then** 1 **else**

    if even(n) **then** f(n/2, x) $\underline{\uparrow 2}$

    else $\underline{x *}$ f(n-1,x)

⟦**mix**⟧ [f,5]

f(5, x) = **if** 5=0 **then** 1 **else**

if even(5) **then** f(5/2, x) ↑ 2

else x * f(5-1,x)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = **if** false **then** 1 **else**

if even(5) **then** f(5/2, x) <u>↑ 2</u>

else <u>x *</u> f(5-1,x)

$[\![\textbf{mix}]\!]$ [f,5]

f(5, x) = **if** even(5) **then** f(5/2, x) ↑ 2
else x * f(5-1,x)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = **if** false **then** f(5/2, x) <u>↑ 2</u>
else <u>x *</u> f(5-1,x)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> f(5-1,x)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> f(4,x)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> (**if** 4=0 **then** 1 **else**

if even(4) **then** f(4/2, x) <u>↑ 2</u>

else <u>x *</u> f(4-1,x))

⟦**mix**⟧ [f,5]

f(5, x) = x * (**if** false **then** 1 **else**
if even(4) **then** f(4/2, x) ↑ 2
else x * f(4-1,x))

⟦**mix**⟧ [f,5]

f(5, x) = x * (**if** even(4) **then** f(4/2, x) ↑ 2
else x * f(4-1,x))

⟦**mix**⟧ [f,5]

f(5, x) = x * (**if** true **then** f(4/2, x) ↑ 2
else x * f(4-1,x))

⟦**mix**⟧ [f,5]

f(5, x) = x * (f(4/2, x) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> (f(2, x) <u>↑ 2</u>)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> ((**if** 2=0 **then** 1 **else**
if even(2) **then** f(2/2, x) <u>↑ 2</u>
else <u>x *</u> f(2-1,x))<u>↑ 2</u>)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> ((**if** false **then** 1 **else**
if even(2) **then** f(2/2, x) <u>↑ 2</u>
else <u>x *</u> f(2-1,x))<u>↑ 2</u>)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((**if** even(2) **then** f(2/2, x) ↑ 2

else x * f(2-1,x))↑ 2)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((**if** true **then** f(2/2, x) ↑ 2
else x * f(2-1,x)) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((f(2/2, x) ↑ 2) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((f(1, x) ↑ 2) ↑ 2)

⟦mix⟧ [f,5]

f(5, x) = x * ((**if** 1=0 **then** 1 **else**
if even(1) **then** f(1/2, x) ↑ 2
else x * f(1-1,x)) ↑ 2) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((**if** false **then** 1 **else**
if even(1) **then** f(1/2, x) ↑ 2
else x * f(1-1,x)) ↑ 2) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((**if** even(1) **then** f(1/2, x) ↑ 2
else x * f(1-1,x)) ↑ 2) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> ((**if** false **then** f(1/2, x) <u>↑ 2</u>
else <u>x *</u> f(1-1,x)) <u>↑ 2)</u> <u>↑ 2</u>)

⟦**mix**⟧ [f,5]

$f(5, \underline{x}) = \underline{x\ *}\ (\ (\underline{x\ *}\ f(1-1,x)\ )\ \underline{\uparrow\ 2)}\ \underline{\uparrow\ 2}\ )$

⟦**mix**⟧ [f,5]

f(5, <u>x</u>) = <u>x *</u> ((<u>x *</u> f(0,x)) <u>↑ 2)</u> <u>↑ 2</u>)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((x * (**if** 0=0 **then** 1 **else**
if even(0) **then** f(0/2, x) ↑ 2
else x * f(0-1,x))) ↑ 2) ↑ 2)

⟦**mix**⟧ [f,5]

f(5, x) = x * ((x * (**if** true **then** 1 **else**
if even(0) **then** f(0/2, x) ↑ 2
else x * f(0-1,x))) ↑ 2) ↑ 2)

$[\![\textbf{mix}]\!]$ [f,5]

$f(5, \underline{x}) = \underline{x\ *}\ (\ (\underline{x\ *}\ (1)\ )\ \underline{\uparrow\ 2)}\ \underline{\uparrow\ 2}\ )$

⟦**mix**⟧ [f,5]

$f(5, \underline{x}) = \underline{x\ *}\ (\ (\underline{x}\ \underline{\uparrow\ 2)}\ \underline{\uparrow\ 2}\ )$

First Futamura Projection

- Partial evaluating interpreter **int** (implemented in language T) w.r.t its input program f (written in language S) compiles f into an equivalent **target** program (written in language T).

target = $[\![$**mix**$]\!]$ [**int**, f]

$[\![\ [\![\mathbf{mix}]\!]\ [\mathbf{int}, f]\ ]\!]_T\ [in] =$

$[\![\mathbf{int}]\!]_T\ [f, in] =$

$[\![f]\!]_S\ [in]$

target = ⟦**mix**⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

code

```
LOAD 10
ADD 20
STORE 30
HALT
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {

}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;

}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];

}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;

}
```

target

target = ⟦**mix**⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];


}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;

}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;
Mem[30] := ACC;

}
```

target

target = ⟦**mix**⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;
Mem[30] := ACC;
ip := 3;

}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;
Mem[30] := ACC;
ip := 3;
return;
}
```

target

target = ⟦mix⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;
Mem[30] := ACC;
ip := 3;
return;
}
```

target

target = ⟦**mix**⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;
Mem[30] := ACC;
ip := 3;
return;
}
```

target

target = ⟦**mix**⟧ [interp, code]

```
interp (P,in) {
ip=0;
Fetch:
if P[ip] is <LOAD x>
    ACC := Mem[x]
else if P[ip] is <ADD,x>
    ACC := ACC+Mem[x]
else if P[ip] is <STORE, x>
    Mem[x] := ACC
else if P[ip] is <HALT>
    return;
else if ...
ip := ip +1;
goto Fetch;
}
```

P

```
LOAD 10
ADD 20
STORE 30
HALT
```

```
interp(in) {
ACC := Mem[10];
ACC := ACC + Mem[20];
Mem[30] := ACC;
return;
}
```

target

Second Futamura Projection

- Partial evaluating **mix** (implemented in T) w.r.t an interpreter **int** (written in T) yields a **compiler** (implemented in T) that translates programs f from language S to language T.

$$\textbf{compiler} = [\![\textbf{mix}]\!]_T\ [\textbf{mix}, \textbf{int}]$$

$$[\![\ [\![\ [\![\textbf{mix}]\!]_T\ [\textbf{mix}, \textbf{int}]\ ]\!]_T\ [f]\ ]\!]_T\ [in]\ ]\!]_T =$$

$$[\![\ [\![\textbf{mix}]\!]_T\ [\textbf{int}, f]\ ]\!]_T\ [in] =$$

$$[\![\textbf{int}]\!]_T\ [f, in] =$$

$$[\![\textbf{f}]\!]_S\ [in]$$

Third Futamura Projection

- Partial evaluating **mix** (implemented in T) w.r.t **mix** (written in T) yields a compiler-compiler **cogen** (implemented in T) that when given an interpreter **int** for a language S returns a compiler from S to T.

$$\textbf{cogen} = [\![\textbf{mix}]\!]_T\ [\textbf{mix}, \textbf{mix}]$$

$$[\![\ [\![\ [\![\ [\![\textbf{mix}]\!]_T\ [\textbf{mix}, \textbf{mix}]\ ]\!]_T\ [\textbf{int}]\ ]\!]_T\ [f]\ ]\!]_T\ [in] =$$

$$[\![\ [\![\ [\![\textbf{mix}]\!]_T\ [\textbf{mix}, \textbf{int}]\ ]\!]_T\ [f]\ ]\!]_T\ [in] =$$

$$[\![\ [\![\textbf{mix}]\!]_T\ [\textbf{int}, f]\ ]\!]_T\ [in] =$$

$$[\![\textbf{int}]\!]_S\ [f, in]]$$

$$[\![f]\!]_S\ [in]]$$