Programs and the Functions they Implement

• Let $f$ be a program written in language $L$
• $f_L$ is the program meaning function for language $L$
• $f_L\left[\text{in}\right]$ is the result of executing program $f$ on input $\text{in}$
  
  output $= f_L\left[\text{in}\right]$

  i.e., output = function $f_L$ applied to argument $\text{in}$
• 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.,
  \[[\text{int}]_L [in_1, in_2] = [\text{in}_1]_S [in_2]\]
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
  – in₁, the first input for program f

• The result of running mix on inputs [in₁, in₂] is a specialized version of program fin₁ that has its first input, in₂, baked in. The program returned by mix, when presented with second input in₂, returns \([f]_S [in₁, in₂]\)

\[
\left[ \left[ \text{mix} \right] \right]_L [f, in₁] \ o_\ S [in₂] = \left[ f \right]_S [in₁, in₂]
\]
Trivial Example

• Let f be \( (\text{lambda} \ (x \ y)(+ \ x \ y)) \)
• Let f2 = \([\text{mix}] \ [f,2]\)
• f2 could be
  \( (\text{lambda}(d) \ (f \ 2 \ d)) \)
  which is
  \( (\text{lambda}(d) \ ((\text{lambda} \ (x \ y)(+ \ x \ y)) \ 2 \ d)) \)
  but this doesn’t count, as partial evaluation is done to speed up computation, not slow it down
• A better f2 would be \( (\text{lambda} \ (y)(+ \ 2 \ y)) \) 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

\[ f(n, x) = \begin{cases} 
  1 & \text{if } n=0 \\
  \text{if even}(n) \text{ then } f(n/2, x) \uparrow 2 & \text{else x * f(n-1,x)} 
\end{cases} \]

• Let f5 = \( [\text{mix}] [f,5] \)

• A good f5 obtained by evaluating the loop at partial-evaluation time would be

\[ f5(x) = x * ((x \uparrow 2) \uparrow 2) \]

• 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

\[
f(n, x) = \text{if } n=0 \text{ then } 1 \text{ else } \\
\quad \text{if even}(n) \text{ then } f(n/2, x) \uparrow 2 \\
\quad \text{else } x * f(n-1, x)
\]

• 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
$$f(n, x) = \begin{cases} 1 & \text{if } n=0 \\ \text{if } \text{even}(n) \text{ then } f(n/2, x)^2 & \\ \text{else } x \times f(n-1, x) & \end{cases}$$
\[ \text{mix} \] \[ f,5 \]

\[ f(5, x) = \text{if } 5=0 \text{ then } 1 \text{ else } \]
\[ \text{if even}(5) \text{ then } f(5/2, x) \uparrow 2 \]
\[ \text{else } x \times f(5-1, x) \]
\[ \text{[mix]} [ f, 5 ] \]

\[
f(5, x) = \begin{cases} 
1 & \text{if false} \\
\text{if even}(5) \text{ then } f(5/2, x) \uparrow 2 & \text{else}
\end{cases}
\]

\[
\text{else } x * f(5-1, x)
\]
\[[\text{mix}] \ [f,5]\]

\[
f(5, x) = \begin{cases} 
& \text{if even}(5) \text{ then } f(5/2, x) \uparrow 2 \\
& \text{else } x \ast f(5-1, x) 
\end{cases}
\]
\[ \text{mix} \] \[ f, 5 \]

\[ f(5, x) = \text{if false then } f(5/2, x) \uparrow 2 \]

\[ \text{else } x \ast f(5-1, x) \]
\[ [\text{mix}] \ [f,5] \]

\[ f(5, x) = x \cdot f(5-1, x) \]
$[\text{mix}] [f, 5]$

\[ f(5, x) = x \times f(4, x) \]
\([\text{mix}] [f, 5]\)

\[f(5, x) = x^* (\text{if } 4=0 \text{ then } 1 \text{ else if even}(4) \text{ then } f(4/2, x) \uparrow^2 \text{ else } x^* f(4-1, x) )\]
\[ \text{mix} \] \[ f,5 \]

\[ f(5, x) = x^* \text{ ( if false then 1 else if even(4) then } f(4/2, x) \uparrow 2 \text{ else } x^* f(4-1, x) ) \]
\[
[f, 5] = [\text{mix}] [f, 5]
\]

\[
f(5, x) = x^* (\text{if even}(4) \text{ then } f(4/2, x) \uparrow 2 \\
\text{else } x^* f(4-1, x))
\]
mix^[f,5]$

\[
f(5, x) = x^* ( \text{if true then } f(4/2, x) \uparrow 2 \\
\text{else } x^* f(4-1,x) )
\]
mix \[ f, 5 \]

\[
f(5, x) = x^* ( f(4/2, x) \uparrow 2 )
\]
½[\textbf{mix}] [f, 5]

\[
f(5, x) = x^* (f(2, x) \uparrow 2)
\]
\[
\text{mix} \ [f, 5] \\
\]
\[
f(5, x) = x^\star ( \text{if } 2=0 \text{ then } 1 \text{ else } \text{if } \text{even}(2) \text{ then } f(2/2, x) \uparrow 2 \text{ else } x^\star f(2-1, x) \uparrow 2 )
\]
\[ \text{mix} \begin{bmatrix} f, 5 \end{bmatrix} \]

\[
f(5, x) = x^* ( \text{if false then 1 else}
\]
\[
\text{if even}(2) \text{ then } f(2/2, x)^{\uparrow 2}
\]
\[
\text{else } x^* f(2-1, x)^{\uparrow 2} )
\]
\[ \text{mix} \{ f, 5 \} \]

\[
f(5, x) = x^* \left( \begin{array}{l}
\text{if even}(2) \text{ then } f(2/2, x) \uparrow^2 \\
\text{else } x^* f(2-1, x) \uparrow^2
\end{array} \right)
\]
\[ \text{mix} \left[ f, 5 \right] \]

\[
f(5, x) = x^* \left( \text{if true then } f(2/2, x) \uparrow 2 \right. \\
\left. \text{else } x^* f(2-1, x) \uparrow 2 \right)\
\]
mix \[ [f,5] \]

\[ f(5, x) = x^* \left( ( f(2/2, x) \uparrow 2 \right) \uparrow 2 ) \]
$\text{mix} \; [f, 5]$

$f(5, x) = x^* ( ( f(1, x)^2)^2 )$
\[
\text{[mix]} \ [f, 5]
\]

\[f(5, x) = x^* \ ( \text{if } 1=0 \ \text{then } 1 \ \text{else} \]
\[ \text{if even}(1) \ \text{then} \ f(1/2, x) \uparrow 2 \]
\[ \text{else} \ x^* \ f(1-1, x) \uparrow 2) \uparrow 2 \) \]
\[ \text{mix} \] \[ f,5 \]

\[ f(5, x) = x^* ( \text{if false then 1 else} \]
\[ \text{if even(1) then } f(1/2, x) \uparrow 2 \]
\[ \text{else } x^* f(1-1,x) \uparrow 2) \uparrow 2 ) \]
mix[f,5] = x^2 \cdot \left( (\text{if even}(1) \text{ then } f(1/2, x) \uparrow^2 \right.
\text{else } x^* f(1-1,x) ) \uparrow^2 \right) \uparrow^2
\[
[mix] \ [f, 5]
\]

\[
f(5, x) = x^* \ ( ( \text{if false then } f(1/2, x) \uparrow 2 \\
\text{else } x^* f(1-1,x) ) \uparrow 2 ) \uparrow 2 \)
\]
\[
\text{mix} \quad [f, 5]
\]

\[
f(5, x) = x^* \left( (x^* f(1-1, x)) \uparrow 2 \right) \uparrow 2
\]
\[ \text{mix} \] \quad [f, 5] \\

\[ f(5, x) = x^* ( (x^* f(0, x) ) ↑ 2 ) ↑ 2 \]
int mix[f, 5] = f(5, x) = x^2 (x^2 (if 0=0 then 1 else if even(0) then f(0/2, x) \uparrow 2 else x^2 f(0-1,x) ) ) \uparrow 2 \uparrow 2 )
\[
\text{mix} \begin{bmatrix} \text{[} f, 5 \text{]} \end{bmatrix}
\]

\[
f(5, x) = x^* ( (x^* (\text{if true then } 1 \text{ else if even}(0) \text{ then } f(0/2, x) \uparrow 2 \\
\text{else } x^* f(0-1, x) \uparrow 2) \uparrow 2) \uparrow 2 )
\]
\[ \text{mix} \] \[ f,5 \]

\[ f(5, x) = x^2 \cdot (x^2 \cdot (1)^2)^2 \]
\( \text{mix} \) \([f, 5]\)

\[
f(5, x) = x^* \left( (x \uparrow 2) \uparrow 2 \right)
\]
**First Futamura Projection**

- Partial evaluating interpreter $\text{int}$ (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 $T$).

$$\text{target} = \llbracket \text{mix} \rrbracket [\text{int}, f]$$

\[
\llbracket \llbracket \text{mix} \rrbracket [\text{int}, f] \rrbracket_T [\text{in}] = \\
\llbracket \text{int} \rrbracket_T [f, \text{in}] = \\
\llbracket f \rrbracket_S [\text{in}] 
\]
target = \texttt{[mix]} [interp, code]

interp (P,in) {
    ip=0;
    Fetch:
    \textbf{if} P[ip] is \langle LOAD x \rangle  
    \hspace{1cm} ACC := Mem[x]
    \textbf{else if} P[ip] is \langle ADD, x \rangle  
    \hspace{1cm} ACC := ACC+Mem[x]
    \textbf{else if} P[ip] is \langle STORE, x \rangle  
    \hspace{1cm} Mem[x] := ACC
    \textbf{else if} P[ip] is \langle HALT \rangle  
    \hspace{1cm} return;
    \textbf{else if} ...  
    ip := ip +1;
    \textbf{goto} Fetch;
}
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;
}

interp(in) {
    }

P
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;
}

interp(in) {
ip=0;

\[
\begin{align*}
P &= \begin{cases}
  &\text{LOAD 10} \\
  &\text{ADD 20} \\
  &\text{STORE 30} \\
  &\text{HALT}
\end{cases}
\end{align*}
\]

\textbf{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;
}

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

P

LOAD 10
ADD 20
STORE 30
HALT

target
\[ \text{target} = \begin{bmatrix} \text{mix} \end{bmatrix} [\text{interp, code}] \]

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

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

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

```
\textbf{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;
}

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

P
LOAD 10
ADD 20
STORE 30
HALT

target
target = \texttt{[mix]} [interp, code]

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

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

P

\begin{tabular}{l}
  LOAD 10 \\
  ADD 20 \\
  STORE 30 \\
  HALT \\
\end{tabular}

\texttt{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;
}

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

P
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;
}

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

P

LOAD 10
ADD 20
STORE 30
HALT

target
target = \texttt{[mix]} [interp, code]

\begin{verbatim}
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;
}
interp(in) {
ip=0;
ACC := Mem[10];
ip := 1;
ACC := ACC + Mem[20];
ip := 2;
ip := 3;
return;
}
\end{verbatim}
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;
}

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

target

P
LOAD 10
ADD 20
STORE 30
HALT
target = \[
\text{mix} \] \text{[interp, code]}

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

interp(in) {
    ip=0;
    ACC := Mem[10];
    ip := 1;
    ACC := ACC + Mem[20];
    ip := 2;
    ip := 3;
    \textbf{return};
}
target = \([\text{mix}]\) [interp, code]

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

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

P
LOAD 10
ADD 20
STORE 30
HALT

target
Second Futamura Projection

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

\[
\text{compiler} = [\text{mix}]_T [\text{mix}, \text{int}]
\]

\[
[ [ [\text{mix}]_T [\text{mix}, \text{int}] ]_T [f] ]_T [\text{in}] ]_T =
\]

\[
[ [\text{mix}]_T [\text{int}, f] ]_T [\text{in}] =
\]

\[
[\text{int}]_T [f, \text{in}] =
\]

\[
[f]_S [\text{in}]
\]
Third Futamura Projection

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

\begin{align*}
\text{cogen} &= \left[ \left[ \text{mix} \right]_T \right] [\text{mix}, \text{mix}] \\
\left[ \left[ \left[ \text{mix} \right]_T \right] [\text{mix}, \text{mix}] \right]_T [\text{int}] &\left[ f \right]_T [\text{in}] = \\
\left[ \left[ \text{mix} \right]_T \right] [\text{mix}, \text{int}] &\left[ f \right]_T [\text{in}] = \\
\left[ \text{mix} \right]_T [\text{int}, f] &\left[ f \right]_T [\text{in}] = \\
[\text{int}]_S [f, \text{in}] & \\
[\text{f}]_S [\text{in}] &
\end{align*}