# **Partial Evaluation**

Eelco Visser

Center for Software Technology

Utrecht University

visser@cs.uu.nl

September 24, 2002

## Partial Evaluation = Program Specialization

Partial evaluation

\_

Specializing a program to it's static (known) inputs

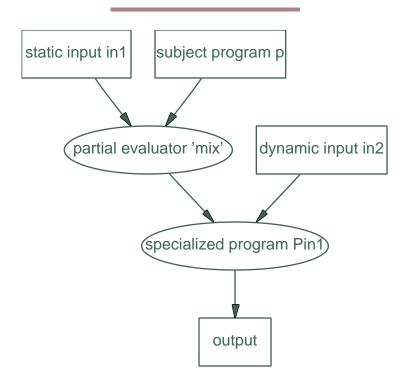
Source: Chapter 1 of Neil Jones, Carsten K. Gomar and Peter Sestoft. *Partial Evaluation and Automatic Program Generation* 

#### Kleene's S-M-N Theorem

There is a primitive recursive function  $\sigma$  such that

$$f(x,y) = \sigma(f,x)(y)$$

# Partial Evaluator (Mix)



```
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then square(power(x, n/2))
  else (x * power(x, n - 1))
```

Tiger

$$\Downarrow$$
 n = 5

Tiger

```
function power5(x : int) : int =
  x * square(square(x))
```

```
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then square(power(x, n/2))
  else (x * power(x, n - 1))
```

Tiger

$$\Downarrow$$
 n = 5

Tiger

```
function power5(x : int) : int =
  x * square(square(x))
```

precompute all expressions involving n

```
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then square(power(x, n/2))
  else (x * power(x, n - 1))
```

Tiger

$$\downarrow$$
 n = 5

Tiger

```
function power5(x : int) : int =
  x * square(square(x))
```

precompute all expressions involving n
unfold recursive call to power

```
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then square(power(x, n/2))
  else (x * power(x, n - 1))
```

Tiger

$$\downarrow$$
 n = 5

Tiger

```
function power5(x : int) : int =
  x * square(square(x))
```

precompute all expressions involving n
unfold recursive call to power
reduce x\*1 to x

- Symbolic computation
  - compute with expressions involving variables (not only values)

- Symbolic computation
  - compute with expressions involving variables (not only values)
- Unfolding
  - replace call with instantiated body of function

- Symbolic computation
  - compute with expressions involving variables (not only values)
- Unfolding
  - replace call with instantiated body of function
- Program point specialization
  - create new version of function, specialized to some arguments
  - = definition + folding

- Definition
  - introduce a new definition, or
  - extend an existing definition

- Definition
  - introduce a new definition, or
  - extend an existing definition
- Folding
  - replace an expression with a function call

- Definition
  - introduce a new definition, or
  - extend an existing definition
- Folding
  - replace an expression with a function call
- Memoization
  - store the result of some computation
  - reproduce the result when computation is needed again

### **Terminology and Notation**

- p : program
- $\bullet$   $[p]_L$ : meaning of program p in language L
- L : implementation language
- S : source language
- T : target language
- ullet  $[\![ \ ]\!]_{\mathbb{L}}$  has type  $D o D^* o D$

$$\mathtt{output} = \llbracket \mathtt{p} \rrbracket_{\mathtt{L}} \left[ \mathtt{in}_1, \mathtt{in}_2, ..., \mathtt{in}_n \right]$$

one stage computation

$$\mathtt{out} = \llbracket \mathtt{p} \rrbracket \; [\mathtt{in1}, \mathtt{in2}]$$

one stage computation

$$\mathtt{out} = \llbracket \mathtt{p} \rrbracket \ [\mathtt{in1}, \mathtt{in2}]$$

two stage computation

$$\begin{aligned} \mathbf{p}_{\text{in1}} &= \llbracket \texttt{mix} \rrbracket \; [\texttt{p}, \texttt{in1}] \\ \text{out} &= \llbracket \mathbf{p}_{\text{in1}} \rrbracket \; \texttt{in2} \end{aligned}$$

one stage computation

$$out = [p][in1, in2]$$

two stage computation

$$p_{\mathtt{in1}} = [\![\mathtt{mix}]\!] \ [p,\mathtt{in1}]$$

$$out = [p_{in1}] in2$$

equational definition of mix

$$[p][in1, in2] = [[mix][p, in1]][in2]$$

one stage computation

$$\mathtt{out} = \llbracket \mathtt{p} \rrbracket \ [\mathtt{in1}, \mathtt{in2}]$$

two stage computation

$$p_{in1} = [mix][p, in1]$$

$$out = [p_{in1}][in2]$$

equational definition of mix

$$[p][in1, in2] = [[mix][p, in1]][in2]$$

different source, target, and implementation languages

$$[\![p]\!]_S [\mathtt{in1},\mathtt{in2}] = [\![[\mathtt{mix}]\!]_L [\![p,\mathtt{in1}]]\!]_T \mathtt{in2}$$

### Speedup

- $t_p(d_1, ..., d_n)$ : time to compute  $[p]_L [d_1, ..., d_n]$
- $-t_p(in1, in2)$ : time to run program
- $-t_{mix}(p, in1)$ : time to specialize p to in1
- $-t_{p_{in1}}(in2)$ : time to run specialized program

#### Speedup

- $-t_p(d_1,...,d_n)$ : time to compute  $\llbracket p \rrbracket_L [d_1,...,d_n]$
- $-t_p(in1, in2)$ : time to run program
- $-t_{mix}(p, in1)$ : time to specialize p to in1
- $-t_{p_{in1}}(in2)$ : time to run specialized program
- specialized program is faster (many applications to in1)

$$\mathtt{t}_{\mathtt{p}_{\mathtt{in1}}}(\mathtt{in2}) < \mathtt{t}_{\mathtt{p}}(\mathtt{in1},\mathtt{in2})$$

### Speedup

- $t_p(d_1, ..., d_n)$ : time to compute  $[p]_L [d_1, ..., d_n]$
- $-t_p(in1, in2)$ : time to run program
- $-t_{mix}(p, in1)$ : time to specialize p to in1
- $-t_{p_{in1}}(in2)$ : time to run specialized program
- specialized program is faster (many applications to in1)

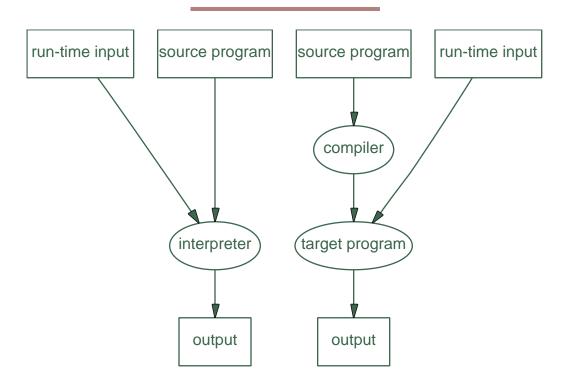
$$\mathtt{t}_{\mathtt{p}_{\mathtt{in1}}}(\mathtt{in2}) < \mathtt{t}_{\mathtt{p}}(\mathtt{in1},\mathtt{in2})$$

specialization and running is faster (one application to in1)

$$t_{mix}(p, in1) + t_{p_{in1}}(in2) < t_{p}(in1, in2)$$

- Efficient and modular solution
  - program adapted to one problem: efficient, but difficult to maintain
  - modular/generic program: easy to maintain and produce different instance, but inefficient
  - partial evaluation: specialize generic program to specific instance

# **Staged Computation**



#### **Interpreters and Compilers**

interpreter int executes a source program

$$\begin{array}{lll} \texttt{output} &=& \llbracket \texttt{source} \rrbracket_{\mathtt{S}} \left[ \texttt{in}_1, ..., \texttt{in}_n \right] \\ &=& \llbracket \texttt{int} \rrbracket_{\mathtt{L}} \left[ \texttt{source}, \texttt{in}_1, ..., \texttt{in}_n \right] \end{array}$$

compiler generates object program from source

$$target = [compiler]_L [source]$$

running target

$$\begin{array}{ll} \texttt{output} &=& \llbracket \texttt{source} \rrbracket_{\mathtt{S}} \left[ \texttt{in}_1, ..., \texttt{in}_n \right] \\ &=& \llbracket \texttt{target} \rrbracket_{\mathtt{T}} \left[ \texttt{in}_1, ..., \texttt{in}_n \right] \end{array}$$

#### **Parser Generators**

generating a parser

$$parser_{grammar} = [parsegen]_{L} [grammar]$$

parsing a string

$$parsetree = [\![parser_{grammar}]\!]_L \ [string]$$

general parsers

$$parsetree = [genparser]_L [grammar, string]$$

# **Generalization: Executable Specifications**

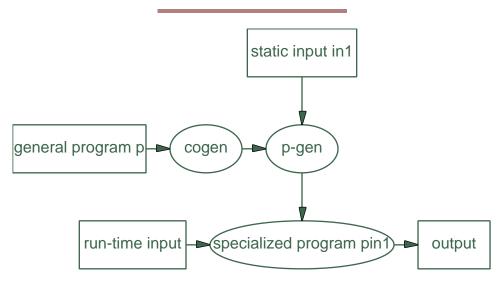
specification executer specexec 'runs' specification spec

$$[\![\mathtt{specexec}]\!][\mathtt{spec},\mathtt{in}_1,...,\mathtt{in}_n]$$

specexec is highly generic program

examples of such executers: int, genparser

### **Program Generator Generation**



$$compiler = [cogen] int$$

parsegen = [cogen] genparser

# **Compiling by Partial Evaluation**

First Futamura Projection (1971): specialize interpreter to source

$$\texttt{target} = [\![\mathtt{mix}]\!] \ [\mathtt{int}, \mathtt{source}]$$

## **Compiling by Partial Evaluation**

First Futamura Projection (1971): specialize interpreter to source

$$target = [mix][int, source]$$

Compiling by partial evaluation yields correct target program:

### **Compiling by Partial Evaluation**

First Futamura Projection (1971): specialize interpreter to source

$$\texttt{target} = \llbracket \texttt{mix} \rrbracket \ [\texttt{int}, \texttt{source}]$$

Compiling by partial evaluation yields correct target program:

- target is in output language of mix
- specialization removes overhead of inspecting source

# **Compiler Generation**

Second Futamura Projection

$$compiler = [mix][mix, int]$$

## **Compiler Generation**

## Second Futamura Projection

$$compiler = [mix][mix, int]$$

#### Correctness

# **Compiler Generator Generation**

Third Futamura Projection

$$cogen = [mix][mix, mix]$$

## **Compiler Generator Generation**

Third Futamura Projection

$$cogen = [mix][mix, mix]$$

Correctness

#### Next

#### Partial evaluation

- What does mix do?
- Online and offline partial evaluation
- Binding time analysis
- Binding time improvents

#### Term rewriting

- Program = Abstract Syntax Tree = Term
- Program Transformation = Modifying AST = Term Rewriting