The Futamura Projections for Compiling and Compiler Generation

Neil D. Jones
DIKU, University of Copenhagen
Outline

✓ Compilers
  ▪ Program specialisation =
    partial evaluation = Kleene’s S-m-n theorem
  ▪ Compiling by specialising an interpreter
  ▪ Futamura projections (self-application put to practical use)
  ▪ Compiler ``bootstrapping`` (... no room in this talk ...)

7/9/08
Universal programs, or self-interpreters

**Definition:** $\text{int} \in L$-programs is a *self-interpreter*, or *universal program* (for $L$) if

$$\llbracket \text{int} \rrbracket (p, d) = \llbracket p \rrbracket (d)$$

for all $p \in L$-programs and $d \in L$-data.
Self-application of a universal program

An immediate consequence of this definition:

\[ \text{int}(\text{int}, (p,d)) = \text{int}(p,d) = p(d) \]

for all \( p \in \text{L-programs} \) and \( d \in \text{L-data} \).

Typically, for some \( a > 10 \) and all \( d \in \text{L-data} \):

\[ \text{time}_{\text{int}}^L(p,d) > a * \text{time}_p^L(d) \]

Consequence:

double interpretation multiplies runtimes... (!)
Compilers

**Notation:** \( \text{comp} \, \varepsilon \, \text{L} \) is a compiler

- **from:** \( \text{Src} \) (source language)
- **to:** \( \text{Tgt} \) (target language)
- **written in:** \( \text{L} \) (implementation language)

**Correctness of** \( \text{comp} \): for all \( \text{src} \in \text{Src} \)-programs and \( \text{in} \in \varepsilon \, \text{Src-data} \):

\[
[[\text{src}]^{\text{Src}}\,\text{(in)}] = [[[[\text{comp}]^{\text{L}}\,(\text{src})]]^{\text{Tgt}}\,(\text{in})]
\]
- progress -

✓ Compilers

✓ **Program specialisation** =

  partial evaluation = Kleene’s *S-m-n theorem*

  - Compiling by *specialising an interpreter*
  - **Futamura projections** (self-application put to practical use)
Normal evaluation:

\[ [\text{in1, in2}] \rightarrow p \rightarrow \text{result} \]
Normal evaluation  Partial Evaluation 1:

\[ [\text{in1}, \text{in2}] \rightarrow \text{p} \]

\[ \text{p} \rightarrow \text{p}_{\text{in1}} \]

\[ \text{in1} \rightarrow \text{specialiser} \]

\[ \text{result} \]
Normal evaluation Partial Evaluation 2:

\[ [\text{in1}, \text{in2}] \rightarrow \text{p} \]

\[ \text{Oval} = \text{data} \]

\[ \text{Rectangle} = \text{program} \]

Note: \( \text{p}_{\text{in1}} \) plays both roles.
Program specialiser `spec`
Specialisation of the `power` function (computes $x^n$)

$$f(n, x) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot f(n - 1, x) & \text{if } \text{odd}(n) \\ f(n/2)^2 & \text{else} \end{cases}$$

$$\text{power}_{13} = f_{13}(x) = x \cdot ((x \cdot (x^2))^2)^2$$
Specialisation of the `power` function (computes $x^n$)

\[ \text{power}_{13} = \begin{cases} 
  f(13,x) = & \text{if } 13 = 0 \text{ then } 1 \\
  & \text{else if } \text{odd}(13) \text{ then } x \times f(13-1,x) \\
  & \text{else } \ldots \\
\end{cases} \]

\[ = \begin{cases} 
  f_{13}(x) = x \times f(12,x) \\
\end{cases} \]

\[ f(12,x) = \begin{cases} 
  & \text{if } 12 = 0 \text{ then } 1 \\
  & \text{else if } \text{odd}(12) \text{ then } \ldots \\
  & \text{else } f(12/2)^2 \\
\end{cases} \]
### Specialisation of ``power´´ function

**power**\(_{13}\) =

\[
\begin{align*}
f(13,x) &= \begin{cases} 
1 & \text{if } 13=0 \\
xf(13-1,x) & \text{if odd}(13) \\
\ldots & \text{else}
\end{cases} \\
\end{align*}
\]

= \[f_{13}(x) = xf(12,x)\]

= \[
\begin{align*}
f(12,x) &= f(6)**2 \\
f(6,x) &= f(3)**2 \\
f(3,x) &= xf(2) \\
f(2,x) &= f(1)**2 \\
f(1,x) &= xf(0) \\
f(0,x) &= 1
\end{align*}
\]
Specialisation of the `power´ function (computes $x^n$)

\[
\text{power} = \begin{align*}
\text{f}(n,x) &= \\
\text{if } n=0 &\text{ then } 1 \\
\text{else if } \text{odd}(n) &\text{ then } x\text{f}(n-1,x) \\
\text{else } &\text{f}(n/2)^{**2}
\end{align*}
\]

\[
\text{power}_{13} = \begin{align*}
\text{f}_{13}(x) &= x\text{f}(n-1,x) \\
&= x((x^2)^{**2})^{**2}
\end{align*}
\]
Binding-time analysis

- static, compute at specialisation time
- dynamic, generate code to compute at run time

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

We know that we will know \( n \) and will not know \( x \), when specialisation begins.
## Binding-time analysis

- **Static**: compute at specialisation time
- **Dynamic**: generate code to compute at run time

\[
f(n, x) =
\begin{align*}
\text{if } n = 0 & \text{ then } 1 \\
\text{else if } \text{odd}(n) & \text{ then } x \cdot f(n - 1, x) \\
\text{else } & f(n/2, x)^2
\end{align*}
\]

\[\text{power} =
\]

If we know \( n \), we can decide whether it is zero.
Binding-time analysis

- static, compute at specialisation time
- dynamic, generate code to compute at run time

\[ f(n, x) = \begin{cases} 
1 & \text{if } n = 0 \\
 x \cdot f(n-1, x) & \text{if odd}(n) \\
f(n/2, x)^2 & \text{else}
\end{cases} \]

power =

-- and whether it is odd,
Binding-time analysis

= static, compute at specialisation time
= dynamic, generate code to compute
at run time

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

-- and we can compute \( n-1 \) or \( n/2 \).
Binding-time analysis

= static, compute at specialisation time
= dynamic, generate code to compute at run time

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

More: all function calls may be unfolded, since \( n \) decreasing at each call and is bounded below.
- progress -

✓ Compilers

✓ Program specialisation =
  partial evaluation = Kleene’s S-m-n theorem

✓ Compiling by specialising an interpreter
  Futamura projections (self-application put to practical use)
The Futamura projections: a preview

In a compiling context, they’re easy to state:

\[
\begin{align*}
\text{target} & := [\text{spec}](\text{int,source}) \\
\text{compiler} & := [\text{spec}](\text{spec,int}) \\
\text{cogen} & := [\text{spec}](\text{spec,spec})
\end{align*}
\]

where \text{int} is an interpreter for some language (call it S), and \text{source} is an S-program.

Real self-application, with practical use...
Compiling by **specialising an interpreter**

**Claim**: One can compile by specialising a (cross-language) interpreter.

\[
\text{target} := \text{[spec]}(\text{int}, \text{program})
\]

This compiles

- **From** the interpreter’s input language
- **To** the specialiser’s output language
An example of an interpreter

The mini-language Norma:

Program syntax:

\[
\begin{align*}
\text{pgm} & ::= [\text{instr}, \ldots, \text{instr}] \\
\text{instr} & ::= \text{X}:=\text{X+1} \mid \text{X}:=\text{X-1} \mid \text{Y}:=\text{Y+1} \mid \text{Y}:=\text{Y-1} \\
& \mid \text{ifX=0goto i} \mid \text{ifY=0goto i} \mid \text{goto i}
\end{align*}
\]

Example program \( p \) : \((\text{input} = \text{X}, \text{output} = \text{Y})\)

\[
[\text{Y}:=\text{Y+1}, \text{Y}:=\text{Y+1}, \text{ifX=0goto 5}, \text{X}:=\text{X-1}, \text{goto 0}]
\]

Example computation by program \( p \) :

\[\lbrack p \rbrack(2) = 6\]
Interpreter "normaint" for Norma

execute(pgm, x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) = case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
  if x = 0 then run(tail(rest), pgmcopy, x, y)
  else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = -- find suffix of pgm starting with
          the lth instruction --
Binding-time analysis of ”normaint”

execute(pgm, x) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) = case head(rest) of
"X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
"X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
"goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
"ifX=0goto l" :
  if x = 0 then run(tail(rest), pgmcopy, x, y)
  else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for ”Y” instructions --
lookup(l, pgm) = -- find suffix of pgm starting with
  the lth instruction --
Binding-time analysis of "normaint"

execute(\texttt{pgm}, \texttt{x}) = run(\texttt{pgm}, \texttt{pgm}, \texttt{x}, 0)
run(\texttt{rest}, \texttt{pgmcopy}, \texttt{x}, \texttt{y}) = \text{ case head} (\text{rest}) \text{ of}
"\texttt{X:=X+1}" : run(\text{tail} (\text{rest}), \texttt{pgmcopy}, \texttt{x+1}, \texttt{y})
"\texttt{X:=X-1}" : run(\text{tail} (\text{rest}), \texttt{pgmcopy}, \texttt{x-1}, \texttt{y})
"\texttt{goto l}" : run(\text{lookup} (l, \texttt{pgmcopy}, \text{rest}), \texttt{pgmcopy}, \texttt{x}, \texttt{y})
"\texttt{ifX=0goto l}" :
  if \texttt{x} = 0 then run(\text{tail} (\text{rest}), \texttt{pgmcopy}, \texttt{x}, \texttt{y})
  else run(\text{lookup} (l, \texttt{pgmcopy}, \text{rest}), \texttt{pgmcopy}, \texttt{x}, \texttt{y})
-- similar for "Y" instructions --
lookup(l, \texttt{pgm}) = -- find suffix of \texttt{pgm} starting with
the lth instruction --
Binding-time analysis of "normaint"

execute(pgm[x]) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) = case head(rest) of
  "X:=X+1": run(tail(rest), pgmcopy, x+1, y)
  "X:=X-1": run(tail(rest), pgmcopy, x-1, y)
  "goto l": run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
  "ifX=0goto l": if x = 0 then run(tail(rest), pgmcopy, x, y)
                    else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = -- find suffix of pgm starting with
               the lth instruction --
Binding-time analysis of "normaint"

execute(pgmx) = run(pgm, pgm, x, 0)
run(rest, pgmcopy, x, y) = case head(rest) of
  "X:=X+1" : run(tail(rest), pgmcopy, x+1, y)
  "X:=X-1" : run(tail(rest), pgmcopy, x-1, y)
  "goto l" : run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
  "ifX=0goto l" :
    if x = 0 then run(tail(rest), pgmcopy, x, y)
    else run(lookup(l, pgmcopy, rest), pgmcopy, x, y)
-- similar for "Y" instructions --
lookup(l, pgm) = - find suffix of pgm starting with
                    the lth instruction -
Binding-time analysis of "normaint"

execute\((\text{pgm}, \text{x})\) = run\((\text{pgm}, \text{pgm}, \text{x}, 0)\)

run\((\text{rest, pgmcopy}, \text{x}, \text{y})\) = \text{case head\((\text{rest})\) of}

"X:=X+1": run\((\text{tail\((\text{rest})\), pgmcopy}, \text{x+1}, \text{y})\)

"X:=X-1": run\((\text{tail\((\text{rest})\), pgmcopy}, \text{x-1}, \text{y})\)

"goto l": run\((\text{lookup\((l, pgmcopy), rest\), pgmcopy}, \text{x}, \text{y})\)

"ifX=0goto l":

\text{if} \text{x} = 0 \text{then run(}\text{tail\((\text{rest})\), pgmcopy}, \text{x}, \text{y})\)

\text{else run(}\text{lookup\((l, pgmcopy), rest\), pgmcopy}, \text{x}, \text{y})\)

-- similar for "Y" instructions --

lookup\((l, pgm)\) = -- find suffix of pgm starting with the lth instruction --
Example of source and target programs

Example source program \( p \):

\[
[ Y := Y + 1, Y := Y + 1, \text{if} X = 0 \text{goto } 5, X := X - 1, \text{goto } 0 ]
\]

Example target program \([\text{spec}](\text{int}, p)\)

\[
\begin{align*}
\text{execute}(x) &= \text{run0}(x, 0) \\
\text{run0}(x, y) &= \text{run2}(x, y + 1 + 1) \\
\text{run2}(x, y) &= \text{if } x = 0 \text{ then } \text{run5}(x, y) \text{ else } \text{run3}(x, y) \\
\text{run3}(x, y) &= \text{run0}(x, y) \\
\text{run5}(x, y) &= y
\end{align*}
\]

Thus: \( p \) is compiled from imperative to functional form.
- progress -

✓ Compilers
✓ Program specialisation =
  partial evaluation = Kleene’s S-m-n theorem
✓ Compiling by specialising an interpreter
✓ The Futamura projections
  (self-application put to practical use)
The **Futamura projections**

**Claim:** specialisation and self-application can
- Compile, given an interpreter and source program
- Generate a compiler, given an interpreter
- Generate a compiler generator (by itself)

**Using:** definitions of specialiser and interpreter:
- \([\text{int}](\text{source}.d) = [\text{source}]^S(d)\)
- \([p](s,d) = [[[\text{spec}](p,s)]](d)\)
Futamura projection 1 is Correct

\[
\begin{align*}
\text{target} & := [\text{spec}](\text{int},\text{source}) \\
\text{compiler} & := [\text{spec}](\text{spec},\text{int}) \\
\text{cogen} & := [\text{spec}](\text{spec},\text{spec})
\end{align*}
\]

To show: \text{target} has same output as \text{source}.

\[
\begin{align*}
\text{out} & = [\text{source}]^S(\text{in}) & \text{Running source S-program} \\
& = [\text{int}](\text{source},\text{in}) & \text{Definition of interpreter} \\
& = [[[\text{spec}](\text{int},\text{source})]](\text{in}) & \text{Def’n of specialiser} \\
& = [\text{target}](\text{in}) & \text{Definition of target}
\end{align*}
\]
Futamura projection 2 is Correct

\[
\begin{align*}
\text{target} & := \text{[spec]}(\text{int}, \text{source}) \\
\text{compiler} & := \text{[spec]}(\text{spec}, \text{int}) \\
\text{cogen} & := \text{[spec]}(\text{spec}, \text{spec})
\end{align*}
\]

To show: \text{compiler} generates target from source.

\[
\begin{align*}
\text{target} & = \text{[spec]}(\text{int}, \text{source}) & \text{Projection 1} \\
& = \text{[[spec] (spec, int)] (source)} & \text{Def’n of specialiser} \\
& = \text{[compiler]}(\text{source}) & \text{Def’n of compiler}
\end{align*}
\]
Futamura projection 3 is Correct

```plaintext
target := [spec](int.source)
compiler := [spec](spec,int)
cogen := [spec](spec,spec)
```

To show: `cogen` generates compiler from `int`.

```
compiler = [spec](spec,int)  # Projection 2
    = [[spec](spec,spec)](int)  # Def'n of specialiser
    = [cogen](int)             # Def'n of cogen
```
Conclusions

- Self-reference is very common in Computer Science
- It mostly (always?) involves programs
- Used in foundations:
  - unsolvability of the halting problem,
  - recursive constructions, Kleene’s recursion theorem, etc.
- Foundational concepts natural in Computer Science:
  - universal function and Kleene’s S-m-n property
- Unexpected but practical applications:
  - compiling by specialising an interpreter
  - compiler generation by self-applying a program specialiser