Partial Evaluation:

Types, Binding Times and Optimal Specialisation

Lecture 2: Partial Evaluation, Compiling, and Compiler Generation

Neil D. Jones
DIKU, University of Copenhagen (prof. emeritus)
Interpreters, etc. all L-programs
and $d, s \in S$-data

Definition \textit{int} is an interpreter \textit{(for S written in L)} if

$$[[p]]^S(d) = [[[\text{int}]]^L(p, d)$$

\textit{comp} is a compiler \textit{(from S to T written in L)} if

$$[[p]]^S(d) = [[[\text{comp}]]^L(p)]^T(d)$$

\textit{spec} is a specialiser \textit{(from S to T written in L)} if

$$[[p]]^S(s, d) = [[[\text{spec}]]^L(p, s)]^T(d)$$

For specialisers: henceforth assume $S = L = T$. 
Specialisation as a staging transformation. Program $p$’s computation is not performed all at once on $(s,d)$, but rather in two stages.

1. A program transformation: given $p$ and $s$, yield as output a specialised program $p_s$.
2. Program $p_s$ is run with the single input $d$. 

*Figure 2.1: Execution: normal; and with a program specialiser.*
WHY RUN A PROGRAM IN TWO STAGES?

- Usually, $\text{time}_{ps}(d) < \text{time}_p(s, d)$, so there is an efficiency payoff when $p_s$ is run on many different $d$’s.

- Sometimes, even $\text{time}_{spec}(p, s) + \text{time}_{ps}(d) < \text{time}_p(s, d)$, so there can be an efficiency payoff even for one run of $p_s$. 
a(m,n) = if m =? 0 then n+1 else
    if n =? 0 then a(m-1,1)
    else a(m-1,a(m,n-1))

Suppose we know m = 2, but n is unknown.

1. **Symbolic evaluation**: eliminate test on m.

   a(2,n) = if n=?0 then a(1,1) else a(1,a(2,n-1))

2. **Unfold call** a(1,1), **then symbolic evaluation**:

   a(1,1) = a(0,a(1,0))

3. **Unfolding call** a(1,0), **then symbolic evaluation**

   a(1,0) = a(0,1) = 1+1 = 2
4. From Steps 2 and 3 we get

\[ a(1,1) = a(0, a(1,0)) = a(0,2) = 3 \] and so from Step 1:

\[ a(2,n) = \text{if } n = 0 \text{ then } 3 \text{ else } a(1,a(2,n-1)) \]

5. Similarly:

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

6. Unfolding calls with \( m=0 \) gives simpler program:

\[ a(2,n) = \text{if } n = 0 \text{ then } 3 \text{ else } a(1,a(2,n-1)) \]
\[ a(1,n) = \text{if } n = 0 \text{ then } 2 \text{ else } a(1,n-1) + 1 \]

7. New specialised program points \( a_1, a_2 \):

\[ a_2(n) = \text{if } n = 0 \text{ then } 3 \text{ else } a_1(a_2(n-1)) \]
\[ a_1(n) = \text{if } n = 0 \text{ then } 2 \text{ else } a_1(n-1) + 1 \]
We show that a partial evaluator can be used to

- **compile** (given interpreter and source program);
- **convert an interpreter into a compiler**; and to
- **generate a compiler generator**.

Discovered by Yoshihiko Futamura in 1971.

**Definition** Suppose \( \text{spec} \) is a partial evaluator, and \( \text{int} \) is an interpreter for some language \( S \) written in \( L \), and \( \text{source} \in S\text{-programs} \).

The **Futamura projections** are the following three definitions of programs \( \text{target} \), \( \text{compiler} \) and \( \text{cogen} \).

1. \( \text{target} := \llbracket \text{spec} \rrbracket(\text{int}, \text{source}) \)
2. \( \text{compiler} := \llbracket \text{spec} \rrbracket(\text{spec}, \text{int}) \)
3. \( \text{cogen} := \llbracket \text{spec} \rrbracket(\text{spec}, \text{spec}) \)
Possible to convert an interpreter into a compiler by partial evaluation. Interesting for several reasons:

- Interpreters are
  - smaller,
  - easier to understand, and
  - easier to debug than compilers.

- An interpreter is a (low-level form of) operational semantics, and so can serve as a definition of a programming language.

- The compiler correctness question is completely avoided, since the compiler will always be faithful to the interpreter from which it was generated.

How to prove the Futamura projections?

Equational reasoning: to show the generated programs behave correctly. (Based on correctness definitions of interpreter, specialiser, compiler.)
A two-input annotated program $p^{\text{ann}} = \begin{align*}
a(m,n) &= \text{if } m = 0 \text{ then } n+1 \text{ else } \\
&\quad \text{if } n = 0 \text{ then } a(m-1,1) \text{ else } \\
&\quad a(m-1, a(m,n-1))
\end{align*}$

Program $p$, specialised to static input $m = 2$:

$p_2 = \begin{align*}
a_2(n) &= \text{if } n=0 \text{ then } a_1(1) \text{ else } a_1(a_2(n-1)) \\
a_1(n) &= \text{if } n=0 \text{ then } a_0(1) \text{ else } a_0(a_1(n-1)) \\
a_0(n) &= n+1
\end{align*}$

*Figure 2.2: Specialising Ackermann’s Function program.*
1. BTA, or Binding-time analysis: Know which program inputs will be known (but not values). Effect: classify operation and function calls in

- “static” : do at specialisation time, or
- “dynamic” : generate code, do at run-time (annotated by underlines above).

2. Specialisation proper: given static input (e.g. \( m = 2 \)), obey annotations, generate residual prog.

- Evaluate all non-underlined expressions;
- Unfold non-underlined calls;
- Generate residual code for underlined expres’ns;
- Generate residual calls for underlined fcn calls.
A “trivial” imperative language:

;; A Norma program works on two registers, x and y,
;; each holding a number (\( n = \text{a list of } n \text{ 1’s} \)).
;; INITIALLY \( x = \text{input} \), \( y = 0 \).
;; AT END: output is \( y \text{’s final value} \).

;; Norma syntax: (only 7 instructions)

;; pgm ::= ( instr* )
;; instr ::= (INC-X) | (DEC-X) | (INC-Y) | (DEC-Y)
;; | (ZERO-X? addr) | (ZERO-Y? addr)
;; | (GOTO addr)
;; addr ::= 1*
Example Norma program

;;; Data: a NORMA program. It computes 2 * x + 2.

((INC-Y) ; 0
(INC-Y) ; 1
(ZERO-X? 1 1 1 1 1 1) ; 2
(INC-Y) ; 3
(INC-Y) ; 4
(DEC-X) ; 5
(GOTO 1 1))) ; 6

( Remarkably, any Turing-computable partial function can be computed by a Norma program. In particular, it is in general undecidable whether a Norma program terminates on a given input \( x \).)
Written in Scheme. See lecture notes (and next slides).

Annotated version:

mark parts of the interpreter as \textit{statically computable}, i.e.,

computable from the interpreted program \( p \) alone, without knowing the run-time input to \( p \)

See lecture notes (and next slides).
Compilation by specialising the interpreter with respect to a known, static, Norma program source that computes $2 \times x + 2$.

Source program (imperative):

```
((INC-Y)
 (INC-Y)
 (ZERO-X? 1 1 1 1 1 1 1)
 (INC-Y)
 (INC-Y)
 (INC-Y)
 (DEC-X)
 (GOTO 1 1))
```
RESULT OF SPECIALISATION

Result, made by UNMIX of specialising Norma-int to source is target = \([\text{spec}](\text{Norma-int,source})\):

\[
\text{(define (execute-$1 x)} \\
\text{\quad (if (pair? x) (run-$1 (cdr x) '(1 1 1 1)) '(1 1)))}
\]

\[
\text{(define (run-$1 x y)} \\
\text{\quad (if (pair? x) (run-$1 (cdr x) '(1 1 ,y)) y))}
\]

This also computes \(2 \times x + 2\); but it is a functional program.

Significant points:

- The programming language has been changed;
- The target program is much faster than the interpreter Norma-int.
- How the target program was obtained:
  - Execute the statically annotated parts of Norma-int\(^{\text{ann}}\)
  - Generate residual program code for the dynamically annotated parts.
The second Futamura projection can also be done by applying \textsc{unmix} to the Norma interpreter. See lecture notes for details.

The third Futamura projection can also done by applying \textsc{unmix} to itself.

\textbf{Second projection} is:

\[
\text{compiler} := \llbracket \text{spec} \rrbracket(\text{spec}, \text{int})
\]

\textbf{Effect:} If \( \text{sint} \in L \) \( L \) then \( \text{compiler} \in S \rightarrow L \)

\textbf{Third projection effect}, loosely:

\[
\llbracket \text{cogen} \rrbracket( L ) = S \rightarrow L
\]
By the Futamura projections there are two different ways to do: program execution, compilation, compiler generation, and compiler generator generation:

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

Exact timings vary according to the design of \text{spec} and \text{int}, and with language L.

A pattern often observed in practical computer experiments:

- each equation’s rightmost run is about 10 times faster than the leftmost.

- Moral: self-application can generate programs that run faster!
DESIRABLE PROPERTIES OF A SPECIALISER

- **Totality of \([\text{spec}]\):**
  \[
  \forall \text{ program } p \ \forall \text{ input } s \ \exists p_s = [\text{spec}](p,s)
  \]

- **Completeness:** \forall \text{ program } p \ \forall \text{ input } s:
  \[\text{[spec]} \text{ does all the } p \text{'s computations that depend on } s \text{ alone.}\]

Alas, you can’t have both at the same time!

- **Suppose** \(p\) **doesn’t use input** \(d\).

- **Completeness requires** \(p_s = [\text{spec}](p,s)\)
  to perform all of \(p\)'s computation on \(s\).

- **Thus** \([\text{spec}]\) **isn’t total:**
  \([\text{spec}](p,s)\) fails to terminate if \([p](s,d)\) fails to terminate.

- **This is a problem:** nobody likes compilers that go into infinite loops!

  Solution in practice: hand annotations to reduce risk of nontermination.

  After annotating \(p\): can specialise it to as many static data \(s\) as desired!
THE CONFLICT BETWEEN TERMINATION AND COMPLETE SPECIALISATION

Standard example: annotated power function:

(define (power x n)
  (if (= n 0) 1 (* x (power x (- n 1))))
)

For \( n = 3 \), expect residual program with all static operations eliminated:

(define (power-3 x) (* x (* x (* x 1))))

But what happens if we try to specialise power to \( n = -2 \)?

Since the source program goes into an infinite loop, the specialiser also loops infinitely.
A tempting way out: specialise $p_s$ less completely in the case that $\lbrack p \rbrack(s, d)$ fails to terminate.

This amounts to solving the halting problem(!).

Some practical specialisers use on-line nontermination checks: monitor the static computations as they are being performed, and force a less thorough specialisation if there’s a risk of nontermination.

If such a strategy can detect all nontermination, it must necessarily be overly conservative in some cases.

WHY?

If the strategy were perfect, it would have solved the halting problem!
Also desirable: the specialiser removes all interpretational overhead. Given a self-interpreter:

\[ \text{sint} \in \mathbb{L} \]

By the first Futamura projection, for all data \( d \)

\[ [p](d) = [\text{sint}_p](d) \]

where \( \text{sint}_p = [\text{spec}](\text{sint}, p) \).

Thus \( \text{sint}_p \) is semantically equivalent to \( p \).

The specialiser has removed all interpretational overhead if \( \text{sint}_p \) is at least as efficient as \( p \).
DEFINITION OF OPTIMAL SPECIALISATION

Definition \textit{spec} is \textbf{optimal} for a self-interpreter \textit{sint} if

\[ \forall p, d, \text{sint}_p = [\text{spec}](\text{sint}, p) \text{ implies } \text{time}_{\text{sint}_p}(d) \leq \text{time}_p(d) \]

\begin{itemize}
  \item This definition is a \textbf{useful quality criterion} in constructing practical evaluators.
  \item For some, specialised program \textit{sint}_p always \textbf{identical} to source program \textit{p} (aside from renaming).
  \item Further, “optimality” is an excellent stepping stone to compiler generation by self-application.
\end{itemize}
The condition just proposed is relative to one particular self-interpreter \texttt{sint}. It could therefore be “cheated” by letting \texttt{spec} be:

\[
\text{spec}(\text{Prog}, S) = \begin{cases} 
\text{if } \text{Prog} = \text{sint} \text{ then } S \\
\text{else Trivial spec’n of Prog to } S;
\end{cases}
\]

On the other hand:

\begin{itemize}
\item it is too much to demand
\item that \texttt{spec} yield optimal specialisations of all possible self-interpreters.
\end{itemize}

Conclusion: “optimality” is

\begin{itemize}
\item pragmatically a useful and meaningful concept, but
\item a concept which mathematically speaking could be improved.
\end{itemize}

This problem has not yet been resolved, and so could be a good research topic.