
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)

INTERPRETATION, COMPILATION, SPECIALISATION

► **Interpreters, etc. all** L -*programs*

► **and** $d, s \in S$ -*data*

Definition int **is an interpreter (for** S **written in** L **) if**

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

comp is a compiler (from S **to** T **written in** L **) if**

$$\llbracket p \rrbracket^S(d) = \llbracket \llbracket \text{comp} \rrbracket^L(p) \rrbracket^T(d)$$

spec is a specialiser (from S **to** T **written in** L **) if**

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

For specialisers: henceforth assume $S = L = T$.

SPECIALISATION

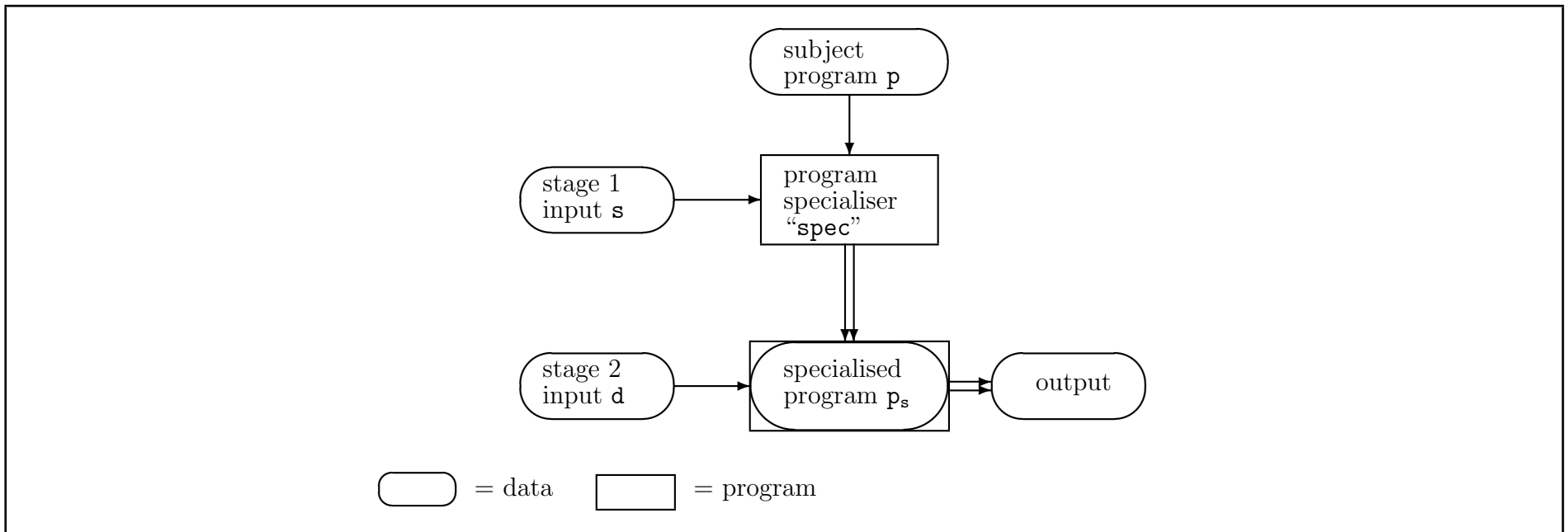


Figure 2.1: A program specialiser.

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 .

WHY RUN A PROGRAM IN TWO STAGES?

- ▶ Usually, $time_{p_s}(d) < time_p(s, d)$, so
there is an efficiency payoff when p_s is run on many different d 's.
- ▶ Sometimes, even $time_{spec}(p, s) + time_{p_s}(d) < time_p(s, d)$, so
there can be an efficiency payoff
even for even for one run of p_s .

SPECIALISING ACKERMANN'S FUNCTION

```
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
```

SPECIALISING ACKERMANN'S FUNCTION (CTD.)

4. From Steps 2 and 3 we get

$$a(1,1) = a(0,a(1,0)) = a(0,2) = 3 \quad \text{and so from Step 1:}$$

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

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 $a1$, $a2$:

$$a2(n) = \text{if } n =? 0 \text{ then } 3 \text{ else } a1(a2(n-1))$$

$$a1(n) = \text{if } n =? 0 \text{ then } 2 \text{ else } a1(n-1)+1$$

THE FUTAMURA PROJECTIONS

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 `spec` is a partial evaluator, and `int` is an interpreter for some language `S` written in `L`, and `source` $\in S\text{-programs}$.

The **Futamura projections** are the following three definitions of programs `target`, `compiler` and `cogen`.

1. `target` := `[[spec]](int, source)`
2. `compiler` := `[[spec]](spec, int)`
3. `cogen` := `[[spec]](spec, spec)`

(We will see the motivation behind these definitions shortly.)

COMPILER GENERATION FROM INTERPRETERS

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.

The fact that

- **we call** these programs `target`, `compiler` and `cogen` **does not mean**
- **they are what the names imply**, i.e., they behave correctly when run.

Next task: show they behave correctly.

FUTAMURA PROJECTION 1: A PARTIAL EVALUATOR CAN COMPILE

Recall functionality of interpreter `int`:

$$\text{out} = \llbracket \text{int} \rrbracket(\text{source}, \text{in})$$

Suppose we want to compute

$$\text{out}_1 = \llbracket \text{int} \rrbracket(\text{source}, \text{in}_1)$$
$$\text{out}_2 = \llbracket \text{int} \rrbracket(\text{source}, \text{in}_2) \dots$$

(run the interpreted source program on several different inputs.) Idea: take

$$\text{target} := \llbracket \text{spec} \rrbracket(\text{int}, \text{source}) (= \text{int}_{\text{source}})$$

Then

$$\begin{aligned} \llbracket \text{target} \rrbracket(\text{in}_i) &= \llbracket \llbracket \text{spec} \rrbracket(\text{int}, \text{source}) \rrbracket(\text{in}_i) \\ &= \llbracket \text{int} \rrbracket(\text{source}, \text{in}_i) \\ &= \text{out}_i \end{aligned}$$

So `target` is indeed a correct target program wrt. `int` and `source`.

MORE ON FUTAMURA PROJECTION 1: A PARTIAL EVALUATOR CAN COMPILE

Output program `target = [[spec]](int, source)` **will be a correctly compiled version of input program** `source` **if**

$$[[source]]^S = [[target]] \quad (= [[target]]^L)$$

Correct compilation verified as follows (`in` and `out` are input and output data of `source`):

<code>out = [[source]]^S(in)</code>	Assumption
<code>= [[int]](source, in)</code>	Interpreter def'n
<code>= [[[[spec]](int, source)]](in)</code>	Specialiser def'n
<code>= [[target]](in)</code>	target def'n

Thus program `target` **deserves its name. Assuming the partial evaluator is correct, this always yields**

**target programs that are correct
with respect to the source programs**

FUTAMURA PROJECTION 2: A PARTIAL EVALUATOR CAN GENERATE A COMPILER

Recall definition of target program target :

$$\text{target} = \llbracket \text{spec} \rrbracket(\text{int}, \text{source})$$

Suppose we want to compute

$$\text{target}_1 = \llbracket \text{spec} \rrbracket(\text{int}, \text{source}_1)$$

$$\text{target}_2 = \llbracket \text{spec} \rrbracket(\text{int}, \text{source}_2)$$

$\vdots$

(That is, compile several different source programs.)

Idea: take

$$\text{compiler} := \llbracket \text{spec} \rrbracket(\text{spec}, \text{int}) (= \text{spec}_{\text{int}})$$

CLAIM: $\llbracket \text{spec} \rrbracket(\text{spec}, \text{int})$ IS A COMPILER

Then

$$\begin{aligned}\llbracket \text{compiler} \rrbracket(\text{source}_i) &= \llbracket \llbracket \text{spec} \rrbracket(\text{spec}, \text{int}) \rrbracket(\text{source}_i) \\ &= \llbracket \text{spec} \rrbracket(\text{int}, \text{source}_i) \\ &= \text{target}_i\end{aligned}$$

So compiler is indeed a correct compiler wrt. int.

The compiler is constructed by **self-application** — using spec to specialise itself to an interpreter.

Hard to understand operationally – but gives good results in practice.

FUTAMURA PROJECTION 3: A PARTIAL EVALUATOR CAN GENERATE A COMPILER GENERATOR

Recall definition of `compiler`:

$$\text{compiler} = \llbracket \text{spec} \rrbracket (\text{spec}, \text{int})$$

Suppose we want to compute

$$\text{compiler}_1 = \llbracket \text{spec} \rrbracket (\text{spec}, \text{int}_1)$$
$$\text{compiler}_2 = \llbracket \text{spec} \rrbracket (\text{spec}, \text{int}_2)$$
$$\vdots$$

(That is, construct compilers from several different interpreters.)

Idea: take

$$\text{cogen} := \llbracket \text{spec} \rrbracket (\text{spec}, \text{spec}) (= \text{spec}_{\text{spec}})$$

CLAIM: $\llbracket \text{spec} \rrbracket(\text{spec}, \text{spec})$ IS A COMPILER

Then

$$\begin{aligned}\llbracket \text{cogen} \rrbracket(\text{int}_i) &= \llbracket \llbracket \text{spec} \rrbracket(\text{spec}, \text{spec}) \rrbracket(\text{int}_i) \\ &= \llbracket \text{spec} \rrbracket(\text{spec}, \text{int}_i) \\ &= \text{compiler}_i\end{aligned}$$

So `cogen` is indeed a correct compiler generator.

`cogen` is constructed by **double self-application** — using `spec` to specialise itself to itself.

Even harder to understand operationally – but also gives good results in practice.

2-PHASE OR OFFLINE PARTIAL EVALUATION

A two-input
annotated
program

$p^{\text{ann}} =$

```
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))
```

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

$p_2 =$

```
a2(n) = if n=0 then a1(1) else a1(a2(n-1))  
a1(n) = if n=0 then a0(1) else a0(a1(n-1))  
a0(n) = n+1
```

Figure 2.2: Specialising Ackermann's Function program.

STAEPS IN 2-PHASE OR OFFLINE PARTIAL EVALUATION

1. BTA, or Binding-time analysis: Know **which program inputs** will be known (but not values). Effect: classify operation and function calls in p
 - ▶ “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.

FIRST FUTAMURA PROJECTION WITH UNMIX

A “trivial” imperative language:

```
;; A Norma program works on two registers, x and y,  
;; each holding a number ( n = a list of n 1's).  
;; INITIALLY x = input, y = 0.  
;; AT END: output is y'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*
```

AN EXAMPLE NORMA PROGRAM

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 1) ; 2
 (INC-Y)                ; 3
 (INC-Y)                ; 4
 (DEC-X)                ; 5
 (GOTO 1 1)))           ; 6
                        ; 7
```

(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 .)

NORMA INTERPRETER

▶ Written in SCHEME. See lecture notes (and next slides).

▶ Annotated version:

mark parts of the interpreter as **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).

SPECIALISATION OF NORMA-INT TO SOURCE

Compilation by specialising the interpreter with respect to a known, static, Norma program source that computes $2 * x + 2$.

Source program (imperative):

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

RESULT OF SPECIALISATION

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

```
(define (execute-$1 x)
  (if (pair? x) (run-$1 (cdr x) '(1 1 1 1)) '(1 1)))
```

```
(define (run-$1 x y)
  (if (pair? x) (run-$1 (cdr x) '(1 1 ,y)) y))
```

This also computes $2 * x + 2$; 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^{ann}
 - Generate residual program code for the dynamically annotated parts.

THE SECOND AND THIRD FUTAMURA PROJECTIONS WITH UNMIX

- ▶ The second Futamura projection can also be done by applying `UNMIX` to the `Norma` interpreter.

See lecture notes for details.

- ▶ The third Futamura projection can also be done by applying `UNMIX` to itself.

SPEEDUPS FROM SELF-APPLICATION

By the Futamura projections there are **two different ways** to do: program execution, compilation, compiler generation, and compiler generator generation:

```
out      := [[int]](source,input) = [[target]](in)
target   := [[spec]](int,source)  = [[compiler]](source)
compiler := [[spec]](spec,int)    = [[cogen]](int)
cogen    := [[spec]](spec,spec)   = [[cogen]](spec)
```

Exact timings vary according to the design of `spec` and `int`, and with language `L`.

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 $\llbracket \text{spec} \rrbracket$:

$$\forall \text{ program } p \ \forall \text{ input } s \ \exists p_s = \llbracket \text{spec} \rrbracket(p, s)$$

- ▶ **Completeness**: $\forall \text{ program } p \ \forall \text{ input } s$:

$\llbracket \text{spec} \rrbracket$ does **all** the p 's computations that depend on s alone.

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

- ▶ Suppose p doesn't use input d .

- ▶ Completeness requires $p_s = \llbracket \text{spec} \rrbracket(p, s)$

to perform all of p 's computation on s .

- ▶ Thus $\llbracket \text{spec} \rrbracket$ isn't total:

$\llbracket \text{spec} \rrbracket(p, s)$ **fails to terminate** if $\llbracket p \rrbracket(s, d)$ 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.

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.

CAN WE ESCAPE THE HORNS OF THE DILEMMA ?

A tempting way out: specialise p_s less completely in the case that $\llbracket p \rrbracket(s, d)$ 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!

OPTIMAL SPECIALISATION

Also desirable: the specialiser removes **all interpretational overhead**. Given a self-interpreter:

$$\text{ sint} \in \boxed{\begin{matrix} L \\ L \end{matrix}}$$

By the first Futamura projection, for all data d

$$\llbracket p \rrbracket(d) = \llbracket \text{sint}_p \rrbracket(d)$$

where $\text{sint}_p = \llbracket \text{spec} \rrbracket(\text{sint}, p)$.

Thus sint_p is semantically equivalent to p .

The specialiser has **removed all interpretational overhead** if sint_p is at least as efficient as p .

DEFINITION OF OPTIMAL SPECIALISATION

Definition `spec` is **optimal** for a self-interpreter `sint` if

$$\forall p, d, \text{sint}_p = \llbracket \text{spec} \rrbracket(\text{sint}, p) \text{ implies } \text{time}_{\text{sint}_p}(d) \leq \text{time}_p(d)$$

- ▶ This definition is a **useful quality criterion** in constructing practical evaluators
- ▶ For some, specialised program `sintp` always **identical** to source program `p` (aside from renaming).
- ▶ Further, “optimality” is an excellent stepping stone to compiler generation by self-application.

A FLY IN THE OINTMENT

The condition just proposed is **relative to one particular self-interpreter** `sint`. It could therefore be “cheated” by letting `spec` be:

```
spec(Prog, S) = if Prog = sint then S
                else Trivial spec'n of Prog to S;
```

On the other hand:

- ▶ it is **too much to demand**
- ▶ that `spec` yield optimal specialisations of **all possible** self-interpreters.

Conclusion: “optimality” is

- ▶ pragmatically a useful and meaningful concept, but
- ▶ a concept which mathematically speaking could be improved.

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