Chapter 4

Partial Evaluation, Compiling, and Compiler Generation

Neil Jones

4.1 Specialization

This section begins by sketching the way that a partial evaluator can work, and then shows the sometimes surprising capabilities of partial evaluation for generating program generators.

Program specialization is a *staging transformation*. Instead of performing the computation of a program `p` all at once on its two inputs `(s,d)`, the computation can be done in two stages. We suppose input `s`, called the *static* input, will be supplied first and that input `d`, called the *dynamic* input, will be supplied later.

In this chapter we assume the partial evaluator `spec` involves one language `L` (so the `L`, `T`, `S` mentioned earlier are all the same). To simplify the notation we will mostly omit `L`, for instance writing $[\![\mathtt{p}]\!](\mathtt{d})$ instead of $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d})$. Thus correctness of `spec` is (adapting from Chapter 3 of these notes):

Definition 4.1.1 Program `spec` is a *specializer* if for any `p` ∈ `L`-*programs* and `s,d` ∈ `L`-*data*

$$[\![\mathtt{p}]\!](\mathtt{s},\mathtt{d}) = [\![[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})]\!](\mathtt{d})$$

The first stage, given `p` and `s`, yields as output a specialized program $\mathtt{p}_\mathtt{s} = [\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})$. In the second stage, program $\mathtt{p}_\mathtt{s}$ is run with the single input `d`—see Figure 3.1[1].

A slightly more complex example: partial evaluation of Ackermann's function. This program computes a function well-known from mathematical logic:

```
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 that `m = 2`, but the value of `n` is unknown. Partial evaluation by hand can be done as follows:

1. *Symbolic evaluation*: for `m=2` the test on `m` can be eliminated, yielding

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

2. *Unfolding* of the call `a(1,1)`, followed by symbolic evaluation yields

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

[1] Notation: data values are in ovals, and programs are in boxes. The specialized program $\mathtt{p}_\mathtt{s}$ is first considered as data and then considered as code, whence it is enclosed in both. Further, single arrows indicate program input data, and double arrows indicate outputs. Thus `spec` has two inputs while $\mathtt{p}_\mathtt{s}$ has only one; and $\mathtt{p}_\mathtt{s}$ is the output of `spec`.

3. Unfolding of the call `a(1,0)` and symbolic evaluation yields

```
a(1,0) = a(0,1) = 1+1 = 2
```

4. From Steps 2 and 3 and symbolic evaluation we get

```
a(1,1) = a(0,a(1,0)) = a(0,2) = 3
```

   and so from Step 1

```
a(2,n) = if n =? 0 then 3 else a(1,a(2,n-1))
```

5. Similar steps yield

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

6. Finally, unfolding the calls with `m=0` gives a simpler program:

```
a(2,n) = if n =? 0 then 3 else a(1,a(2,n-1))
a(1,n) = if n =? 0 then 2 else a(1,n-1) + 1
```

7. The final transformation is *program point specialization*: New functions `a1, a2` are defined so `a2(n)` = `a(2,n)` and `a1(n)` = `a(1,n)`, yielding a less general program that is about twice as fast as the original:

```
a2(n)  =  if n =? 0 then 3 else a1(a2(n-1))
a1(n)  =  if n =? 0 then 2 else a1(n-1)+1
```

Partial evaluation is an automated scheme to realize transformations such as these.

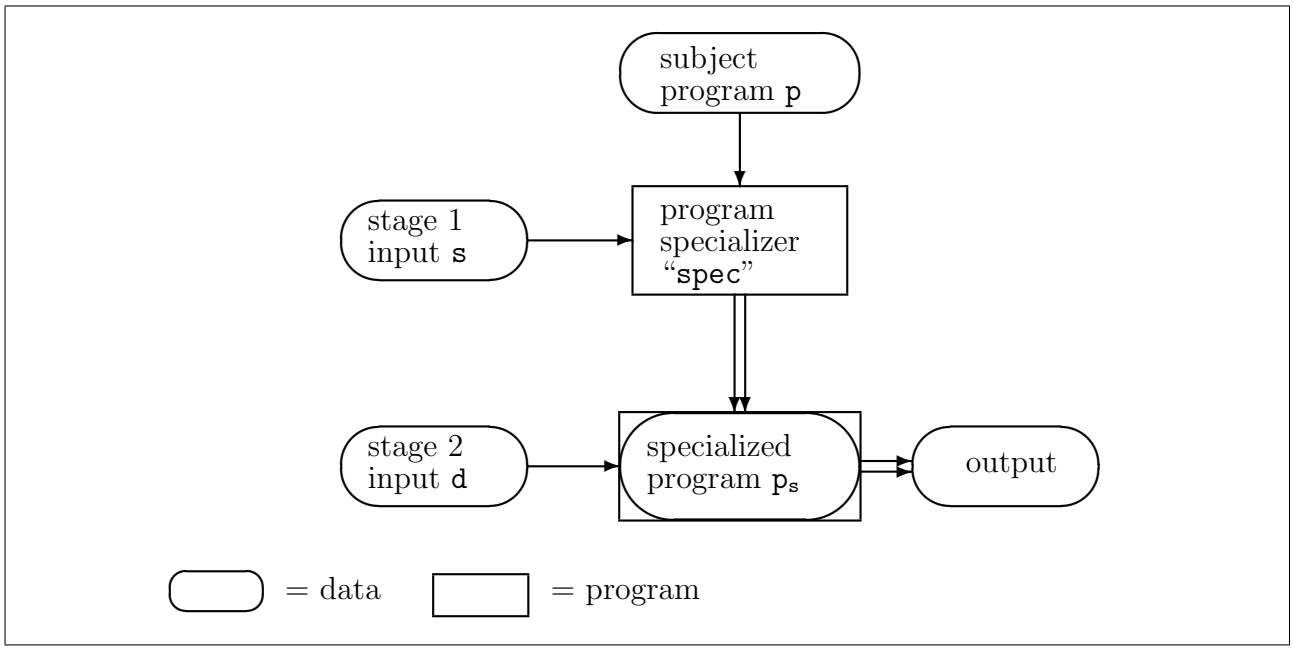


Figure 4.1: A program specializer.

4.2 The Futamura projections

We show now that a partial evaluator can be used to *compile* (if given an interpreter and a source program in the interpreted language); to *convert an interpreter into a compiler*; and to *generate a compiler generator*. The results are called the Futamura projections since they were discovered by Yoshihiko Futamura in 1971 [Fut71].

For now we concentrate on correctness; later discussions concern efficiency.

Definition 4.2.1 Suppose `spec` is a partial evaluator, and `int` is an interpreter for some language `S` written in `L`, and `source` ∈ `S`-*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)
```

The fact that we have *called* these programs `target`, `compiler` and `cogen` does not mean that they are what the names imply, i.e., that they behave correctly when run. The next three sections prove that they deserve their names, using the definitions of interpreter and compiler from Chapter 3.

4.2.1 Futamura projection 1: a partial evaluator can compile

Output program `target` will be a correctly compiled version of input program `source` if $[\![\texttt{source}]\!]^{\texttt{S}} = [\![\texttt{target}]\!]$ $(= [\![\texttt{target}]\!]^{\texttt{L}})$. Correct compilation can be verified as follows, where `in` and `out` are input and output data of `source`:

`out`	=	$[\![\texttt{source}]\!]^{\texttt{S}}(\texttt{in})$	Assumption
	=	$[\![\texttt{int}]\!](\texttt{source,in})$	Definition of an interpreter
	=	$[\![[\![\texttt{spec}]\!](\texttt{int,source})]\!](\texttt{in})$	Definition of a specializer
	=	$[\![\texttt{target}]\!](\texttt{in})$	Definition of `target`

Thus program `target` deserves its name.

The first projection shows that one can *compile* source programs from a new language `S` into the output language of the specializer, provided that an interpreter for `S` is given in the input language of the specializer. Assuming the partial evaluator is correct, this always yields target programs that are correct with respect to the source programs from which they were compiled.

4.2.2 Futamura projection 2: a partial evaluator can generate a compiler

Output program `compiler` will be a correct compiler from source language `S` to the target language `T` if $[\![\texttt{compiler}]\!](\texttt{source}) = \texttt{target}$ for any `source` and `target` related as above. Correctness of the alleged compiler compilation can be verified as follows:

`target`	=	$[\![\texttt{spec}]\!](\texttt{int,source})$	First Futamura projection
	=	$[\![[\![\texttt{spec}]\!](\texttt{spec,int})]\!](\texttt{source})$	Definition of a specializer
	=	$[\![\texttt{compiler}]\!](\texttt{source})$	Definition of `compiler`

Thus program `compiler` also deserves its name.

The second projection shows that one can *generate an* `S` *to* `L` *compiler* written in `L`, provided that an interpreter for `S` written in `L` is given, and that the specializer is written in its own

input language. Assuming the partial evaluator is correct, by the reasoning of Section 4.2.1 the generated compiler always yields target programs that are correct with respect to any given source programs.

The compiler works by generating specialized versions of interpreter `int`. The compiler is constructed by self-application — using `spec` to specialize itself. Constructing a compiler this way is hard to understand operationally. But it gives good results in practice, and usually faster compilation than by the first Futamura projection.

4.2.3 Futamura projection 3: a partial evaluator can generate a compiler generator

Finally, we show that `cogen` is a *compiler generator*: a program that transforms interpreters into compilers. Verification is again straightforward:

`compiler`	=	⟦`spec`⟧(`spec,int`)	Second Futamura projection
	=	⟦⟦`spec`⟧(`spec,spec`)⟧(`int`)	Definition of a specializer
	=	⟦`cogen`⟧(`int`)	Definition of `cogen`

Thus program `cogen` also deserves its name.

The compilers so produced are versions of `spec` itself, specialized to various interpreters. Construction of `cogen` involves a double self-application that is even harder to understand intuitively than for the second projection, but also gives good results in practice.

While the verifications above by equational reasoning are straightforward, it is far from clear what their pragmatic consequences are. Answers to these questions form the bulk of the book [JGS93].

Generating extensions

The idea above can be used for more than just compiler generation. Concretely, let `p` be a two-input program, and define

`p-gen` := ⟦`spec`⟧(`spec,p`)

Program `p-gen` is called the *generating extension* of `p`, and has the property that, when applied to a static input `s` to `p`, will directly yield the result $\texttt{p}_\texttt{s}$ of specializing `p` to `s`. Verification is straightforward as follows:

$\texttt{p}_\texttt{s}$	=	⟦`spec`⟧(`p,s`)	Definition of $\texttt{p}_\texttt{s}$
	=	⟦⟦`spec`⟧(`spec,p`)⟧(`s`)	Definition of a specializer
	=	⟦`p-gen`⟧(`s`)	Definition of `p-gen`

Equation `compiler` = ⟦`spec`⟧(`spec,interpreter`) becomes: `compiler` = `interpreter-gen`. In other words:

> *The generating extension of an interpreter is a compiler.*

The generating extension of `spec` is `cogen`

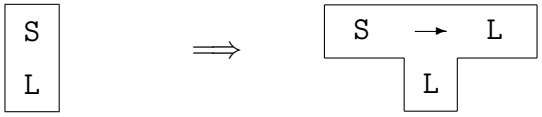
The following equations are also easily verified from the Definition of a specializer:

⟦`p`⟧ (`s,d`)	=	⟦⟦`spec`⟧ (`p,s`) ⟧ (`d`) = ... = ⟦⟦⟦`cogen`⟧ (`p`) ⟧ (`s`) ⟧ (`d`)
`p-gen`	=	⟦`cogen`⟧ (`p`)
`cogen`	=	⟦`cogen`⟧ (`spec`)

The first sums up the essential property of `cogen`, the second shows that `cogen` produces generating extensions, and the third shows that `cogen` can produce itself as output (Exercise 4.2.)

Why do compiler generation?

The effect of running `cogen` can be described diagrammatically:



This is interesting for several practical reasons:

- Interpreters are usually 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, assuming the semantics of `L` is solidly understood.
- The question of compiler correctness is completely avoided, since the compiler will always be faithful to the interpreter from which it was generated.

4.3 Speedups from specialization

This approach has proven its value in practice. See [JGS93] for some concrete speedup factors (often between 3 and 10 times faster). To give a more complete picture, we need to discuss two sets of running times:

1. Target program execution versus interpretation:

$$time_{\texttt{int}}(\texttt{source}, \texttt{d}) \quad \text{versus} \quad time_{\text{int}_{\texttt{source}}}(\texttt{d})$$

2. Target program execution plus specialization versus interpretation:

$$time_{\texttt{int}}(\texttt{source}, \texttt{d}) \quad \text{versus} \quad time_{\text{int}_{\texttt{source}}}(\texttt{d}) + time_{\texttt{spec}}(\texttt{int}, \texttt{source})$$

If program `int` is to be executed only once on input `d`, then comparison 2 is the most fair, since it accounts for what amounts to a form of "compile time." If, however, the specialized program $\texttt{int}_{\texttt{source}}$ is to be run often (e.g. as in typical compilation situations), then comparison 1 is more fair since the savings gained by running $\texttt{int}_{\texttt{source}}$ instead of `int` will, in the long term, outweigh specialization time, even if $\texttt{int}_{\texttt{source}}$ is only slightly faster than `int`.

As mentioned before, compiled programs nearly always run faster than interpreted ones, and the same holds for programs output by the first Futamura projection.

4.4 How specialization can be done

Suppose program `p` expects input `(s,d)` and we know what `s` but not `d` will be. Intuitively, specialization is done by performing those of `p`'s calculations that depend only on `s`, and by generating code for those calculations that depend on the as yet unavailable input `d`. A partial evaluator thus performs a mixture of execution and code generation actions — the reason Ershov called the process "mixed computation" [Ers82], hence the generically used name `mix` for a partial evaluator (which we call `spec`). Its output is often called the *residual program*, the term indicating that it is comprised of operations that could not be performed during specialization.

4.4.1 An example in more detail

For a simple but illustrative example, we will show how the program for Ackermann's function seen earlier can automatically be specialized to various values of its first parameter. Ackermann's function is useless for practical computation, but an excellent vehicle to illustrate the main partial evaluation techniques quite simply. See Figure 4.2 for an example. Note that execution of the specialized program p_2 uses *less than half as many* arithmetic operations as the original.

Computing `a(2,n)` involves recursive evaluations of `a(m,n)` for `m` = 0, 1 and 2, and various values of `n`. The partial evaluator can evaluate expressions `m=0` and `m-1` for the needed values of `m`, and function calls of form `a(m-1,...)` can be unfolded (i.e., replaced by the right side of the recursive definition above, after the appropriate substitutions).

More generally, three main partial evaluation techniques are well known from program transformation: *symbolic computation*, *unfolding function calls*, and *program point specialization*. Program point specialization was used in the Ackermann example to create specialized versions `a0`, `a1`, `a2` of the function `a`.

A two-input annotated program p^{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`, specialized 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 4.2: Specialization of a Program for Ackermann's Function.

Sketch of an off-line partial evaluator.

We assume given an **annotated program** p^{ann}. This consists of:

1. A first-order functional program `p` of form

```
f1(s,d)    =  expression1  (* s,d are static & dynamic inputs, resp. *)
g(u,v,...) =  expression2
...
h(r,s,...) =  expressionm
```

2. Annotations that mark every function parameter, operation, test, and function call as either *eliminable*: to be performed/computed/unfolded during specialization, or *residual*: generate program text to appear in the specialized program.

   (Annotations as "residual" are given by underlines in Figure 4.2.)

The annotations in p^{ann} serve to guide the specializer's actions. The parameters of any definition of a function `f` will be partitioned into those which are *static* and the rest, which are *dynamic*. For instance `m` is static and `n` is dynamic in the Ackermann example.

Form of a specialized program. The specialized program will consist of definitions of *specialized functions* $\texttt{g}_{\texttt{staticvalues}}$. Each of these corresponds to a pair `(g, staticvalues)` where `g` is defined in the original program and `staticvalues` is a tuple consisting of values for all the static parameters of `g`. The parameters of function $\texttt{g}_{\texttt{staticvalues}}$ in the specialized program will be the remaining, dynamic, parameters of `g`.

Example: in the specialized Ackermann program, function `a2` corresponds to the pair `(a, m=2)` and has one residual parameter `n`.

4.4.2 Annotated programs and off-line partial evaluation

An *off-line* partial evaluator works in two phases:

1. **BTA, or Binding-time analysis** to do the annotation: Given information as to *which* program inputs will be known (but not what their values are), the BTA classifies every operation and function call in `p` as

   - "static" : can be evaluated/performed during specialization, or
   - "dynamic" : must appear in the residual program, to be evaluated/performed during run-time.

   In Figure 4.2, dynamic operations and calls are underlined.

2. **Specialization proper:** given the values of the static input parameters ($\texttt{m} = 2$ above), the annotations are obeyed, resulting in generation of a residual program.

The interpretation of the annotations in Figure 4.2 is extremely simple:

- *Evaluate* all non-underlined expressions;
- *unfold at specialization time* all non-underlined function calls;
- *generate residual code* for all underlined expressions; and
- *generate residual function calls* for all underlined function calls.

4.5 The first Futamura projection with UNMIX

Consider a trivial imperative language with this description:

```
;; A Norma program works on two registers, x and y, each holding a
;; number (number n represented as a list of n 1's). The program input
;; is used to initialize x, and y is initialized with 0. The output is
;; y's final value. The allowed instructions include jumps (unconditional
;; and conditional) and increment/decrement instructions for x and y.
;;
;; Norma syntax:
;;
;;   pgm   ::= ( instr* )
;;   instr ::= (INC-X) | (DEC-X) | (INC-Y) | (DEC-Y)
;;           |  (ZERO-X? addr)    |  (ZERO-Y? addr)  |  (GOTO addr)
;;   addr  ::= 1*
```

A `Norma` program to be executed: Input, and the current values of registers `x`, `y` are represented as *unary or base 1 numbers*. Thus `x` = 3 is represented by the length-3 list `(1 1 1)`. As the following program shows, labels are represented by the same device: the final `(GOTO 1 1)` causes transfer to instruction number 2 (the test `ZERO-X?...`).

```
;; Data: a NORMA program.  It computes 2 * x + 2.
  ((INC-Y)
   (INC-Y)
   (ZERO-X? 1 1 1 1 1 1 1)
   (INC-Y)
   (INC-Y)
   (DEC-X)
   (GOTO 1 1)))
```

A simple interpreter `Norma-int` written in SCHEME.

In the code below the function `execute`, given the program and the initial value of `x`, calls `run`. In a call `(run pgtail prog x y)`, parameters `x` and `y` are the current values of the two registers as unary numbers. The call from `run` thus sets `x` to the outside input and `y` to 0, represented by the empty list. Parameter `prog` is always equal to the `Norma` program being interpreted. The current "control point" is represented by a suffix of `prog` called `pgtail`. Its first component is the next instruction to be executed. Thus the initial call to `run` has `pgtail` = `prog`, indicating that execution begins with the first instruction in `prog`.

```
(define (run pgtail prog x y)
 (if (atom? pgtail)    ;;  answer = y if there are no Norma instructions
  y                    ;;      left to execute, else dispatch on syntax

 (let ((instr (car pgtail)) (rest (cdr pgtail)))   ;; first instruction

 (if (atom? instr)
     (cons 'ERROR-invalid-syntax: pgtail)          ;; bad syntax

     (let ((op (car instr)) (arg (cdr instr)))
     (if (equal? op 'INC-X)                          ;; increment x
               (run rest prog (cons '1 x) y)
     (if (equal? op 'DEC-X)                          ;; decrement x
               (run rest prog (cdr x) y)
     (if (equal? op 'ZERO-X?)                        ;; jump if x=0
               (if (pair? x) (run rest prog x y)
                             (run (jump prog arg) prog x y))
     (if (equal? op 'INC-Y)                          ;; increment y
               (run rest prog x (cons '1 y))
     (if (equal? op 'DEC-Y)                          ;; decrement y
               (run rest prog x (cdr y))
     (if (equal? op 'ZERO-Y?)                        ;; jump if y=0
               (if (pair? y) (run rest prog x y)
                             (run (jump prog arg) prog x y))

     (if (equal? op 'GOTO) (run (jump prog arg) prog x y) ;; goto jump

   (cons 'ERROR-bad-instruction: instr))))))))))))  ;; bad instruction
```

```
(define (jump prg dest)     ;; find instruction number "dest" in program
 (if (null? dest)
     prg
     (jump (cdr prg) (cdr dest))))

(define (execute prog x) (run prog prog x (generalize '())))
```

The `generalize` can be ignored for now, as it has no effect on execution.[2]

Annotation of `Norma-int`: The first step of specialization, as described in Section 4.4.2, is binding-time analysis to construct the annotated program. For this the static data is not needed; only its "specialization pattern." The next step is to running UNMIX as decribed in the notes titled *Programs As Data Objects*. For this, UNMIX is to be given specialization pattern `sd` since `prog` is static and `x` is dynamic. Then UNMIX produces `Norma-int`$^{\texttt{ann}}$, the following annotated program:

```
(execute run)
((execute (prog) (x) =  (call run (prog prog) (x (static '()))))

(run (pgtail prog) (x y) =
     (ifs (atom? pgtail) y
                          (call run2 ((car pgtail) pgtail prog) (x y))))

(run2 (instr pgtail prog) (x y) =
  (ifs (atom? instr)
       (static (cons 'ERROR-invalid-syntax: pgtail)
       (call run1 ((car instr) pgtail prog instr) (x y)))))

(run1 (op pgtail prog instr) (x y)  =
  (ifs (equal? op 'INC-X)
       (call run ((cdr pgtail) prog) ((cons (static '1) x) y))
  (ifs (equal? op 'DEC-X)
       (call run ((cdr pgtail) prog) ((cdr x) y))
  (ifs (equal? op 'ZERO-X?)
       (if  (pair? x)
            (call  run ((cdr pgtail)                 prog) (x y))
            (rcall run ((call jump prog (cdr instr)) prog) (x y)))
  (ifs (equal? op 'INC-Y)
       (call run ((cdr pgtail) prog) (x (cons (static '1) y)))
  (ifs (equal? op 'DEC-Y)
       (call run ((cdr pgtail) prog) (x (cdr y)))
  (ifs (equal? op 'ZERO-Y?)
       (if  (pair? y)
            (call  run ((cdr pgtail)                 prog) (x y))
            (rcall run ((call jump prog (cdr instr)) prog) (x y)))
  (ifs (equal? op 'GOTO)
       (rcall run ((call jump prog (cdr instr)) prog) (x y))
  (static (cons 'ERROR-bad-instruction: instr)))))))))))
```

[2]It is a user-supplied binding-time annotation that forces `y` to be dynamic. The reason is to prevent infinite specialization .

```
(jump (prg dest) = (ifs (null? dest)
                        prg
                        (call jump (cdr prg) (cdr dest)))))
```

Explanations concerning the annotated programs produced by UNMIX.

The annotated program has exacly the same structure as the original, except for a variety of annotations:

- A function definition whose source program form is:

  ```
  (define f x1 x2 ... xn) expression)
  ```

  will have its parameter list been split apart, into one list `(s1 s2...sm)` of static parameters followed by a list `(d1 d2...dn)` of dynamic parameters. Thus the definition takes this form in $\texttt{p}^{\texttt{ann}}$:

  ```
  (f (s1 s2...sm) (d1 d2...dn) expression)
  ```

  For example function `run` has two static parameters `(pgtail prog)` and two dynamic parameters `(x y)`. Function `jump` has static arguments only, so it omits `(d1 d2...dn)`.

- In calls, static and dynamic argument lists are split in the same way. For example, in `run2` the call `(call run1 ((car instr) pgtail prog instr) (x y))` has two static arguments: `(car instr)` and `pgtail`; and two dynamic arguments: `x` and `y`.

- Operations appearing in static argument lists are evidently static, and so need no explicit annotations. An example: expression `(car pgtail)` in the `run` definition's call

  `(call run2 ((car pgtail) pgtail prog) (x y))`.

- Similarly, operations appearing in dynamic argument lists are evidently dynamic, and so need not be marked. An example is `(cdr x)` in the call from the `DEC-X` case of `run1`.

- Finally, some operations have been explicitly classified as "static". These appear only in dynamic expressions, and are marked by suffix `s`. All such expressions or tests are computable at specialization time from the static data given to the specializer.

  In $\texttt{Norma-int}^{\texttt{ann}}$, they are only the tests `(ifs e0 e1 e2)` that realize the "dispatch on syntax". In each case, `e0` is static and `e1`, `e2` are dynamic.

- Each residual function call is to appear in the residual program. Such calls are marked as `(rcall name ...)`. These are in effect the "underlined function calls" as used in Section 4.4.2. Any remaining function calls will be performed at specialization time, i.e., the call will be unfolded in-line by the specializer.

  In $\texttt{Norma-int}^{\texttt{ann}}$, the "underlined", i.e., residual, function calls are the ones in the `GOTO` and `ZERO-X` and `ZERO-Y` cases that realize "back jumps" to earlier points in the given `Norma` source program. (Technical point: these must be made residual to keep the specializer from attempting to produce an infinitely large specialized program.)

Specialization proper of `Norma-int` to `source`: Compilation is done by specializing the interpreter with respect to a known, static, `Norma` program. Recall that the static data `source` is the `Norma` program that computes $2 * x + 2$:

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

The result `target` = ⟦`spec`⟧(`Norma-int,source`) of specializing `Norma-int` to `source` is the following SCHEME code produced by UNMIX. It also computes $2 * x + 2$, but is a functional program. The main point: it is much faster than the interpreter `Norma-int`.

```
(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))
```

The target code was generated by executing the annotated interpreter `Norma-int`$^{\texttt{ann}}$. As described in Section 4.4.2, the statically annotated parts of `Norma-int`$^{\texttt{ann}}$ are executed at specialization time, and the dynamically annotated parts are used to generate residual program code.

4.6 The second Futamura projection with UNMIX

We now study the structure of the UNMIX-generated compiler

`compiler := ⟦spec⟧(spec,interpreter)`

for the interpreter `Norma-int` seen above. The listings below, beyond `Norma-int` itself, were obtained by hand-editing UNMIX output files.[3]

The heart of the compiler: generation of residual code from a `Norma` source program. In these functions, `xcode` is the residual expression for `Norma` variable `X`, and similarly for `Y`. Parameters `prog`, `pgtail`, `prg`, `dest` are all exactly as in the `Norma-int`. (Naturally, since all are static.)

Function `pe2` tests to see whether there exists more `Norma` source text to be compiled. If not, then `ycode` is the residual code. If so, it calls function `pe3` with the first `Norma` instruction as an extra parameter. If this is an atom it is illegal syntax and this is signaled. Otherwise, function `pe4` is called with the operation as an extra parameter.

Function `pe4` does *the compile-time dispatch on* `Norma` *syntax:* parameter `instr` is the `Norma` instruction to be compiled, and `op` is its operation code. Note how simple, direct and logical the code below is for the 7 possible instruction forms.

Function `pe4` makes liberal use of SCHEME's "backquote" notation (see the manual for a detailed explanation). A simple example: the following generates a residual `if`-expression, whose test, then- and else-branches are obtained by evaluating expressions `e1,e2,e3`:

[3]The editing only consisted of name changes. It was necessary for understandability, since machine-produced code is full of uninformative mechanically generated parameter names.

```
   `(if ,e1 ,e2 ,e3)
```

This can be written without backquote, but is more cumbersome:

```
   (cons 'if (cons e1 (cons e2 (cons e3 '()))))
```

Code generation functions in the compiler:

```
(define (pe2 pgtail prog xcode ycode)
  (if (atom? pgtail)
      ycode
      (pe3 (car pgtail) pgtail prog xcode ycode)))


(define (pe3 instr pgtail prog xcode ycode)
  (if (atom? instr)
       ``(ERROR-invalid-syntax: ,instr)
       (pe4  (car instr) pgtail prog instr xcode ycode)))


(define (pe4 op pgtail prog instr xcode ycode)
(cond
  ((equal? op 'INC-X) (pe2 (cdr pgtail) prog `(cons '1 ,xcode) ycode))
  ((equal? op 'DEC-X) (pe2 (cdr pgtail) prog `(cdr ,xcode)     ycode))

  ((equal? op 'ZERO-X?)
   `(if (pair? ,xcode)
        ,(pe2 (cdr pgtail) prog xcode ycode)
        (call (run ,(jump prog (cdr instr)) ,prog) ,xcode ,ycode)))

  ((equal? op 'INC-Y) (pe2 (cdr pgtail) prog xcode `(cons '1 ,ycode)))
  ((equal? op 'DEC-Y) (pe2 (cdr pgtail) prog xcode `(cdr ,ycode)))

  ((equal? op 'ZERO-Y?)
    `(if (pair? ,ycode)
         ,(pe2 (cdr pgtail) prog xcode ycode)
         (call (run ,(jump prog (cdr instr)) ,prog) ,xcode ,ycode)))

  ((equal? op 'GOTO)
    `(call (run ,(jump prog (cdr instr)) ,prog) ,xcode ,ycode))

  (else ``(ERROR-bad-instruction: ,instr))))


(define (jump prg dest)
  (if (null? dest)  prg
                    (jump (cdr prg) (cdr dest))))
```

What has been omitted.

The code above is right as far as it goes; but it only shows how the UNMIX-generated compiler generates code from `Norma` commands – and not how it generates the function definitions appearing in the residual program, for example the definitions of `execute-$1` and `run-$1` at the end of Section 4.5.

Rather than present complex code extracted from the compiler, we just explain the underlying ideas in a nontechnical way, for the interpreter `Norma-int`:

1. The specialized program has definitions of specialized functions of form $\texttt{g}_{\texttt{staticvalues}}$ where `g` is defined in the input program (for now, `Norma-int`), and `staticvalues` is a tuple consisting of values for all the static parameters of `g`.

2. The parameters of function $\texttt{g}_{\texttt{staticvalues}}$ in the specialized program will be the remaining, dynamic, parameters of `g`.

3. For the `Norma-int`, the initial specialized function is $\texttt{execute}_{\texttt{source}}\texttt{(x)}$ where static input `source` is the `Norma` program being compiled.

4. The only residual calls `(rcall ...)` in the annotated interpreter are to `run`, which has `pgtail,source` as static parameters and `x,y` as dynamic parameters.

5. Thus all the specialized functions in `target` (other than the start function $\texttt{execute}_{\texttt{source}}$) will have form $\texttt{run}_{\texttt{pgtail,source}}\texttt{(x y)}$ where `pgtail` is a suffix of the `Norma` program `source` being compiled.

6. All calls to `jump` are unfolded at compile time. Furthermore, not all calls to `run` cause residual calls to be generated, i.e., some calls to `run` are also unfolded.

4.7 Speedups from self-application

A variety of partial evaluators generating efficient specialized programs have been constructed. By the easy equational reasoning seen above (based only on the definitions of specializer, interpreter, and compiler), program execution, compilation, compiler generation, and compiler generator generation can each be done in two different ways:

$$\begin{array}{lcccl}
\texttt{out} & := & [\![\texttt{int}]\!]\texttt{(source,input)} & = & [\![\texttt{target}]\!]\texttt{(input)} \\
\texttt{target} & := & [\![\texttt{spec}]\!]\texttt{(int,source)} & = & [\![\texttt{compiler}]\!]\texttt{(source)} \\
\texttt{compiler} & := & [\![\texttt{spec}]\!]\texttt{(spec,int)} & = & [\![\texttt{cogen}]\!]\texttt{(int)} \\
\texttt{cogen} & := & [\![\texttt{spec}]\!]\texttt{(spec,spec)} & = & [\![\texttt{cogen}]\!]\texttt{(spec)}
\end{array}$$

The exact timings vary according to the design of `spec` and `int`, and with the implementation language `L`. We have often observed in practical computer experiments [JGS93] that each equation's *rightmost run is about 10 times faster than the leftmost.* Moral: self-application can generate programs that run faster!

4.8 Metaprogramming without order-of-magnitude loss of efficiency

The right side of Figure 4.3 illustrates graphically that partial evaluation can substantially reduce the cost of the multiple levels of interpretation mentioned earlier.

A literal interpretation of Figure 4.3 would involve writing two partial evaluators, one for `L1` and one for `L0`. Fortunately there is an alternative approach using only *one* partial evaluator, for `L0`. For concreteness let `p2` be an `L2`-program, and let `in`, `out` be representative input and output data. Then

$$\texttt{out} = [\![\texttt{int}_0^1]\!]^{\texttt{L0}}(\texttt{int}_1^2\texttt{,(p2,in)})$$

One may construct an interpreter for `L2` written in `L0` as follows:

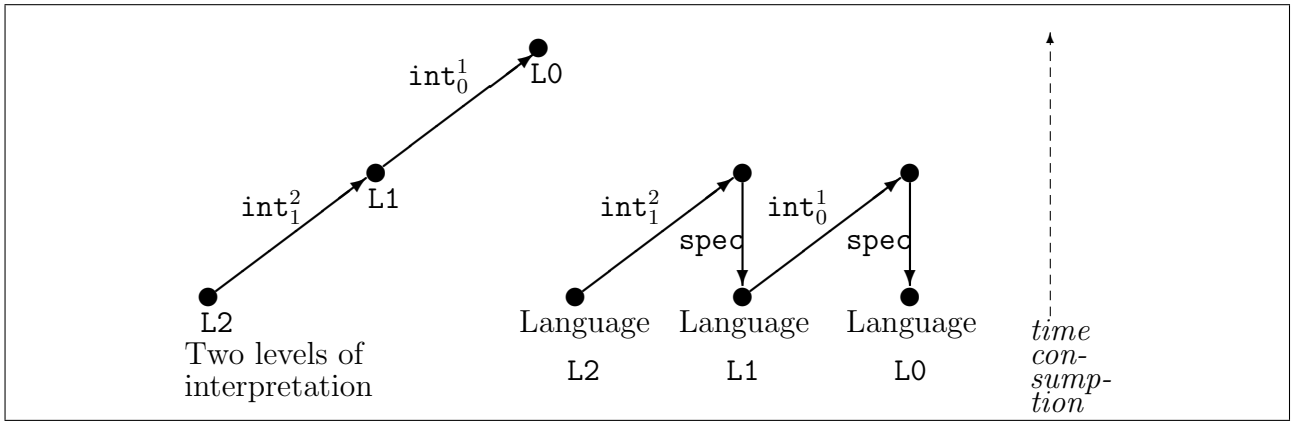


Figure 4.3: Overhead introduction and elimination.

$$\texttt{int}_0^2 := [\![\texttt{spec}]\!]^{\texttt{L0}}(\texttt{int}_0^1, \texttt{int}_1^2) \quad \text{satisfying}$$
$$\texttt{out} = [\![\texttt{int}_0^2]\!]^{\texttt{L0}}(\texttt{p2}, \texttt{in})$$

By partial evaluation of $\texttt{int}_0^2$, any L2-programs can be compiled to an equivalent L0-program. Better still, one may construct a compiler from L2 into L0 by

$$\texttt{comp}_0^2 := [\![\texttt{cogen}]\!]^{\texttt{L0}}(\texttt{int}_0^2)$$

The net effect is that metaprogramming may be used without order–of–magnitude loss of efficiency. The development above, though conceptually complex, has actually been realized in practice by partial evaluation, and yields substantial efficiency gains ([Dyb91], [Jør92]).

4.9 Desirable properties of a specializer

A conflict: totality versus computational completeness

It is clearly desirable that specialization function $[\![\texttt{spec}]\!]$ be total, so *every* program p and partial input s leads to a defined output $\texttt{p}_\texttt{s} = [\![\texttt{spec}]\!](\texttt{p},\texttt{s})$.

Computational completeness

The significant speedups seen in the examples above naturally lead to another demand: that given program p and partial data s, *all* of p's computations that depend only on its partial input s will be performed.

Unfortunately this is in conflict with the desire that $[\![\texttt{spec}]\!]$ be total. Suppose, for example, that program p's computations are independent of its second input d, and that $[\![\texttt{p}]\!]$ is a partial function (i.e., the run $[\![\texttt{p}]\!](\texttt{s},\texttt{d})$ may fail to terminate for some inputs).

Then computational completeness would require $[\![\texttt{spec}]\!](\texttt{p},\texttt{s})$ to do *all* of p's computation on s, so it would also fail to terminate whenever $[\![\texttt{p}]\!](\texttt{s},\texttt{d})$ fails to terminate. This is a problem, since nobody likes compilers or other program transformers that sometimes go into infinite loops!

A typical example which is difficult to specialize nontrivially without having the specializer fail to terminate is indicated by the program fragment

```
if     complex-but-always-true-condition-with-unavailable-input-d
then   X := nil
else   while true do S := cons S S;
```

One cannot reasonably expect the specializer to determine whether the condition will always be true. A specializer aiming at computational completeness will likely attempt to specialize both branches of the `while` loop, leading to nontermination at specialization time.

A tempting way out is to allow $\mathtt{p_s}$ to be less completely specialized in the case that $[\![\mathtt{p}]\!](\mathtt{s,d})$ fails to terminate, e.g. to produce a trivial specialization in just this case. This is, however, impossible in full generality as it would require *solving the halting problem*(!).

Some practical specializers make use of run-time nontermination checks that monitor the static computations as they are being performed, and force a less thorough specialization whenever there seems to be a risk of nontermination. Such strategies, if capable of detecting all nontermination, must necessarily be overly conservative in some cases; for if perfect, they would have solved the halting problem.

Optimality

It is desirable that the specializer be "optimal" when used for compiling, meaning that `spec` removes *all interpretational overhead*. This can be made somewhat more precise, given a self-interpreter `sint`:

$$\mathtt{sint} \in \boxed{\begin{array}{c}\mathtt{L}\\ \mathtt{L}\end{array}}$$

By definition of interpreter and specialization (or by the first Futamura projection), for every data value `d`

$$[\![\mathtt{p}]\!](\mathtt{d}) = [\![\mathtt{sint_p}]\!](\mathtt{d})$$

where $\mathtt{sint_p} = [\![\mathtt{spec}]\!](\mathtt{sint,p})$. Thus program $\mathtt{sint_p}$ is semantically equivalent to `p`. One could reasonably say that the specializer has *removed all interpretational overhead* in case $\mathtt{sint_p}$ is at least as efficient as `p`. We elevate this into a definition:

Definition 4.9.1 Program specializer `spec` is *optimal* for a self-interpreter `sint` in case for every program `p` and data `d`, if $\mathtt{sint_p} = [\![\mathtt{spec}]\!](\mathtt{sint,p})$ then

$$time_{\mathtt{sint_p}}(\mathtt{d}) \leq time_{\mathtt{p}}(\mathtt{d})$$

This definition of "optimality" has proven itself very useful in constructing practical evaluators [JGS93]. For several of these, the specialized program $\mathtt{sint_p}$ is *identical up to variable renaming* to the source program `p`. Further, achieving optimality in this sense has shown itself to be an excellent stepping stone toward achieving successful and satisfactory compiler generation by self-application.

An open problem. Unfortunately there is a fly in the ointment. The condition just proposed is a definition *relative to one particular self-interpreter* `sint`. It could therefore be "cheated" by letting `spec` have the following structure:

```
spec(Program, S) = if Program = sint then S
                   else  a trivial specialization of Program to S;
```

On the other hand, it would be too much to demand that `spec` yield optimal specializations of *all possible* self-interpreters. Conclusion: the concept of "optimality" is pragmatically a good one, but one which mathematically speaking is unsatisfactory. This problem has not been resolved at the time of writing, and so could be a good research topic.

4.10 Exercises

4.1 Explain informally the results claimed in Section 4.7. For example, why should compiling by the run `target` := ⟦`compiler`⟧(`source`) be faster than compiling by the run `target` := ⟦`spec`⟧(`int,source`)?

4.2 Prove the following:

1. ⟦`cogen`⟧ `(p)` is the generating extension of `p`.
2. ⟦`p`⟧ `(s,d)` = ⟦⟦⟦`cogen`⟧ `(p)` ⟧ `(s)` ⟧ `(d)`
3. `cogen` = ⟦`cogen`⟧ `(spec)` .

4.3 By hand: construct a generating extension of the `zip` program from Note set 3.

4.4 Use UNMIX to construct a generating extension of the `zip` program from Note set 3. Examine the code, and describe how it works.

4.5 (Referring to Section 4.6.) Explain which calls to `run` in the interpreter `Norma-int` will cause residual calls to be generated. Does this have implications about the size of a target program?

Is the size of the target program always linear in the size of the source `Norma` program from which it was derived? Justify your answer.

4.6 Explain the need for `generalize` in the `Norma` interpreter, i.e., justify Footnote 2. (Show what undesirable effect would happen if the `generalize` were omitted.)

4.7 A *property list*, or *plist* for short, is a data structure similar to an association list (as described in Exercise 2.2), except that the keys and values are stored in alternate list positions:

$$(key_1\ value_1\ key_2\ value_2\ \ldots\ key_n\ value_n)\,.$$

(You may have encountered plists if you've programmed in Emacs Lisp.) The following questions all deal with looking up keys in plists:

a. Define a Scheme function `pget`, such that the call `(pget pl k d)`, where `pl` is a plist, returns the value corresponding to the first occurrence of `k` as a key in `pl`, or the value `d` if `k` does not occur as a key in `pl`. (You may assume that `pl` is in fact a well-formed plist, and in particular contains an even number of elements.) Demonstrate your function on a couple of representative argument triples.

b. Use UNMIX to specialize your function `pget` to the arguments

$$pl = \texttt{(red roed green groen blue blaa)}, d = \texttt{sort}\,.$$

Show the resulting specialized function, and demonstrate it on a couple of representative keys.

c. Define a Scheme function `pget-gen`, such that the call `(pget-gen pl d)` returns a Scheme program defining a function `pget-r`, such that for any `k`, `(pget-r k)` returns the same result as `(pget pl k d)`. `pget-r` should eliminate as many of the static computations of `pget` as possible; for example, the following (correct, but trivial) generating extension,

```
(define (pget-gen pl d)
   `(define (pget-r k) (pget ',pl k ',d)))
```

would not qualify. Demonstrate your `pget-gen` on the argument pair from part (b), showing both the text of the generated program, and the result of running this program on a few keys.

You may use UNMIX's cogen on `pget` for inspiration, but you will probably find it easier to think about what `pget-gen` should be doing if you try to define it from scratch yourself.

d. Even though `pget-gen` performs all the static computations it can, the definition of `pget-r` it outputs may still not be "optimal" in an informal sense, as demonstrated by taking

$$pl = \texttt{(yes ja no nej yes jo maybe maaske)}, d = \texttt{ved-ikke}\,.$$

Show the output of your `pget-gen` on this data, and explain why the generated program is not as good as one might have hoped for.

e. [Optional] Define an improved variant of `pget-gen`, called `pget-gen-opt`, that does not suffer from the problem identified in part (d). Demonstrate your `pget-gen-opt` on the example above.