Partial Evaluation:
Types, Binding Times and Optimal Specialisation

Neil D. Jones
DIKU, University of Copenhagen (prof. emeritus)

1. Lecture 1: Programs as Data Objects.
3. Lecture 3: The Types Involved in Partial evaluation.

Source: course DAT V Programming Languages, University of Copenhagen
These talks relate to high level implementation of programming languages
(not semantics, not transformation)

Combines two things. Theory: well-defined concepts and
Practice: must be executable on the computer.

Self-reference will be used constructively (programs will be run on themselves as inputs)

A way to use types in connection with program transformation and generation
OUTLINE: PROGRAMS AS DATA OBJECTS

▶ Notations and definitions
  • Programming languages
  • Interpreters
  • Compilers
  • Specialisers

▶ Time costs:
  • Interpretation overhead
  • Compiling versus interpretation
  • Effect of double interpretation
  • A dream: automatically generate compilers from interpreters

▶ Self-interpretation (of a subset of SCHEME)

▶ Partial evaluation systems
WHAT IS A PROGRAMMING LANGUAGE $L$?

A programming language $L$ assigns meanings or effects to entire programs. The meaning of program $p$ in language $L$ written as $[[p]]^L$, for example

$$[[p]]^L(1) = 2$$

Language $L = \text{the semantic function on entire programs}$:

$$[[ - ]]^L : \text{Programs} \rightarrow \text{Meanings}$$

(Omit $L$ if it is understood from context.)

Example:

$$[[ \text{read } X; \text{ write } X + 1 ]]^L : \mathcal{Z} \rightarrow \mathcal{Z}$$

where $\mathcal{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$. 
Programs are not the same as functions!!

We distinguish consistently (though many do not). For an example, the LISP or SCHEME program

\[
p = (\text{lambda}(X)(+ X 1))
\]

is just a string of symbols.

Its meaning $\mathbb{LISP}[p]$ is the mathematical successor function. It satisfies:

\[
\mathbb{LISP}[p](x) \overset{\text{def}}{=} x + 1
\]

for any $x \in \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
WHAT IS A POSSIBLE MEANING OF A PROGRAM?

A few instances:

1. **Computation of a function**: The meaning of program $p$ is a function

   $$[p]^L : \text{Input} \rightarrow \text{Output}$$

   *Meanings* is the set of partial functions defined by $\text{Input} \rightarrow \text{Output}$.

2. **Stream processing**: $p$ interactively reads a finite or infinite input stream $(a_1, a_2, \ldots)$, and produces a stream of outputs $(b_1, b_2, \ldots)$.

   In this case

   $$[p]^L : \text{Input}^{\omega} \rightarrow \text{Output}^{\omega}$$

3. **Communication**: $[p]^L$ specifies how the process specified by $p$

   may interact with other processes. (details omitted)

4. **Effects**: running $p$ may access data from a *global database*, and update its contents. (details omitted)

In these notes: meanings are always input-output functions.
Definition A **programming language** $L$ consists of

- **Two sets:**
  - $L$-programs
  - $L$-data

and

- **A semantic function**

\[
\llbracket \bullet \rrbracket^L : L\text{-programs} \rightarrow (L\text{-data} \rightarrow L\text{-data})
\]

It associates with every $L$-program $p \in L\text{-programs}$ a corresponding partial input-output function

\[
\llbracket p \rrbracket^L : L\text{-data} \rightarrow L\text{-data}
\]
EXAMPLE OF A TINY PROGRAMMING LANGUAGE $L$: POLISH SUFFIX PROGRAMS

Used in some pocket calculators, and sometimes inside compilers. $L$ contains

- **Two sets:**
  - $L$-data = $\mathcal{N} = \{0, 1, 2, \ldots\}$
  - $L$-programs = $\{(I_1 I_2 \ldots I_n) \mid I_i \in \{X, +, *\}\}$

**Example program:** $(X \ X + X *)$

- **Semantic function** for Polish suffix programs
  
  $\llbracket \bullet \rrbracket^L : L$-programs $\rightarrow (\mathcal{N} \rightarrow \mathcal{N})$

  It associates with every $L$-program $p$ an input-output function
  
  $\llbracket p \rrbracket^L : \mathcal{N} \rightarrow \mathcal{N}$

  **Example:**

  $\llbracket (X \ X + X *) \rrbracket^L(7) = 14 \times 7 = 98$
\([\bullet]^L\) is a mathematical object.

Not useful for executing real programs.

(A machine implements only one language in its hardware.)

Interesting if there is an implementation that behaves like \([\bullet]^L\).

Interpreters and compilers:

allow a program written in one language to be executed in another.
OUTLINE: PROGRAMS AS DATA OBJECTS

- Notations and definitions
  - Programming languages
  - Interpreters
  - Compilers
  - Specialisers

- Time costs:
  - Interpretation overhead
  - Compiling versus interpretation
  - Effect of double interpretation
  - A dream: automatically generate compilers from interpreters

- Self-interpretation of a subset of SCHEME

- Partial evaluation systems
How to obtain/invent/implement a new programming language (call it $S$)?

Straightforward, if we already have sufficiently expressive existing language $L$ (and maybe a target language $T$).

- Two ways: **compilation** and **interpretation**:
- Essential: compiler and interpreter must **preserve semantics**.
- Impossible unless **all** of the new language $S$’s semantic effects are expressible in $L$ (or $T$).
Notations and definitions

- Programming languages
- Interpreters
- Compilers
- Specialisers

Time costs:

- Interpretation overhead
- Compiling versus interpretation
- Effect of double interpretation
- A dream: automatically generate compilers from interpreters

Self-interpretation of a subset of SCHEME

Partial evaluation systems
Two programming languages:

- An implementation language $L$, and
- A source language $S$.

Definition  An $L$-program $\text{int}$ is an interpreter of $S$ written in $L$ if for all $p \in S$-programs and $d \in S$-data:

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

Given an $S$-programs and an interpreter $\text{int}$ for source language $S$ written in $L$, then we can execute the $S$-program.

Language constraints:

$$S$-programs $\times$ $S$-data $\subseteq$ $L$-data$$

$$S$-data $\subseteq$ $L$-data$$

Programs are data objects!
; Polish suffix program to compute \( y := x \times x + x \) : \((X X \ast X +)\)
; Program works on natural number stack "st". Output = stack top.
; Three instructions:
; 
; \( X \) : push input on stack: \... \Rightarrow \text{in} \... 
; 
; \(+\) : add top two, push: \(a b\) \... \Rightarrow \(a+b\)...
; 
; \(*\) : multiply top two, push: \(a b\) \... \Rightarrow \(a*b\)...

(define (execute p in) (run p ’() in)) ; stack is initially empty
(define (run p st in)
   (if (empty? p) (head st) ; empty program: execution done
       (if (=?(head p) ’X) (run (tail p) (cons in st) in) ; push in onto st
            (let
                ( (st-1st (head st)) ; stack top
                  (st-2nd (head (tail st))) ; stack next-to-top
                  (st-rest (tail (tail st))) ; stack remainder
               )
                (if (=?(head p) ’+) (run (tail p) (cons (+ st-1st st-2nd) st-rest) in)
                  (if (=?(head p) ’*) (run (tail p) (cons (* st-1st st-2nd) st-rest) in)
                      ’SYNTAX_ERROR )))))))
WHAT DOES AN INTERPRETER DO?

In practice:

▸ Look at each instruction in \( p \) (the current instruction)

▸ Perform the action it specifies

The interpretation equation doesn’t specify how to do interpretation, just its net effect:

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

Interpretation

▸ reduces the problem of implementing \( S \)

▸ to the problem of implementing \( L \)
For all $p \in S$-programs and $d \in S$-data:

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

Meaning of “=” is that on every input $d$ one of these happens:

- Both sides terminate, and give the same result; or
- both terminate with the same error; or
- both loop (fail to terminate)

Some writers are fussy, and write, for example, $\doteq$ to indicate partial equality, e.g.

$$7 \doteq 7 \text{ and } 3 + 4 \doteq 7 \text{ and } 1/0 \doteq 2/0$$

We won’t emphasise termination and so just use $=$
Diagrammatic notation:

- Data value
- Program to be run
- Data input to program
- Data output from program

A 1-step run of program \( p \)
Assume program $p$ computes an input-output function.

**Definition** An interpreter for programming language $S$ written in language $L$ is an $L$-program $int$ such that for any $S$-program $s$ and input $in$:

$$[[p]]^S(d) = [[int]]^L(p, d)$$

**Interpretation:** 1 step
OUTLINE: PROGRAMS AS DATA OBJECTS

► Notations and definitions
  ▪ Programming languages
  ▪ Interpreters
  ▪ **Compilers**
  ▪ Specialisers

► Time costs:
  ▪ Interpretation overhead
  ▪ Compiling versus interpretation
  ▪ Effect of double interpretation
  ▪ A dream: automatically generate compilers from interpreters

► Self-interpretation of a subset of **SCHEME**

► Partial evaluation systems
A compiler translates \textit{S}-programs into \textit{T}-programs.

A “compiler from \textit{S} to \textit{T} written in \textit{L}” is an \textit{L}-program \texttt{comp}.

the target program for the source program \textit{source} is

\[
\text{target} = \texttt{[\text{comp}]^L(source)}
\]

\textbf{Correctness:} the target \textit{T}-program \textit{target} must have the same meaning as the source program \textit{source}. Symbolically

\[
\texttt{[source]^S} = \texttt{[target]^T}
\]

Written as one equation, this must hold for all \textit{S}-programs \textit{source}:

\[
\texttt{[source]^S} = \texttt{[[\text{comp}]^L(source)]^T}
\]
Repeating:

Definition **An L-program** \( \text{comp} \) **is a compiler from** \( S \) **to** \( T \) **if for every** source \( \in S\text{-programs} \) **and** \( d \in S\text{-data}, \)

\[
\left\lfloor \left[ \text{source} \right] \right\rfloor^S(d) = \left\lfloor \left[ \text{comp}]^L(\text{source}) \right\rfloor^T(d)
\]

**Language constraints:**

\( S\text{-programs} \subseteq L\text{-data}, T\text{-programs} \subseteq L\text{-data}, S\text{-data} = T\text{-data}. \)

- No relationship between \( S\text{-data} \) and \( L\text{-data}. \)
- **The compiler** \( \text{comp} \) **is written in language** \( L. \)
Compilation: 2 steps
AN EXAMPLE OF COMPILATION

S: source language = Polish suffix programs
T: target language = SCHEME programs
L: compiler implementation language: unspecified

A compiler translates S-programs into T-programs. An example:

A source POLISH-program

(X X * X +)

A corresponding target SCHEME-program

(define (evaluate in) (+ (* in in) in))

Correctness: target program has the same meaning as source program (?

[source]polish = [target]scheme

Both compute

f(in) = in * in + in

(stated in math, not a programming language.)
OUTLINE: PROGRAMS AS DATA OBJECTS

- Notations and definitions
  - Programming languages
  - Interpreters
  - Compilers
  - Specialisers

- Time costs:
  - Interpretation overhead
  - Compiling versus interpretation
  - Effect of double interpretation
  - A dream: automatically generate compilers from interpreters

- Self-interpretation of a subset of SCHEME

- Partial evaluation systems
Notational conventions:

- **Interpreters, interpreters, specialisers are all** $L$-programs
- **Equations hold for all** $p \in S$-programs
- **and for all** $d, s \in S$-data

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

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

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

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

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

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

Definition An L-program spec is a specialiser of S written in L if for every s, d ∈ S-data,

\[[p]_S^S(s, d) = [[[[spec]]_L^L(p, s)]_T^T](d)\]

Often: the three languages are the same: S = T = L. From now on we use only:

▶ one programming language, so omit it from notation, and write

\[[p](d)\]

instead of \[[p]^L(d)\]

▶ This gives:

\[[p](s, d) = [[[[spec]]_L(p, s)]](d)\]
A two-input program

\[ p = \]
\[ f(n,x) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{else}
\end{cases} 
\]
\[ \text{else } x \ast f(n-1,x) \]

Program \( p \), specialised to static input \( n = 5 \):

\[ p_5 = \]
\[ f_5(x) = x \ast (x \ast (x \ast (x \ast (x \ast 1)))) \]

\( p_5 \) runs faster. **Savings:**

- No operations \( n = 0, n - 1 \)
- No function calls
SPECIALISATION IS 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 $d$. 
PROGRAM SPECIALISATION = PARTIAL EVALUATION

subject program $p$

stage 1 input $s$

program specialiser spec

stage 2 input $d$

specialised program $p_s$

output

= data

= program
Specialisation equation:

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

Partial evaluation: a form of specialisation that optimises program \( p \) with respect to \( s \). Two phases:

- **Analysis**: What depends only on \( s \)?
- **Specialisation**:
  - Simplify what depends only on \( s \) (computations by \( p \)).
  - Construct a \( T \) program for the rest.
1. `spec`, when run on a program `p` and a known value `s` of its first input,
2. yields as output a “specialised” program: `p_s`.
3. `p_s`, if run on `p`’s remaining input `d`,
4. yields the same results that `p` would yield on all of its input.

Practically important: if `p` is

- run frequently with the same value of `s`,
- but with many different values of `d`.

It can even happen that: it may be faster to

- first construct `p_s` from `p` and `s`, and
- then apply `p_s` to `p`’s remaining input data `d`.

rather than to compute `[[p]](s, d)` all at once.
OUTLINE: PROGRAMS AS DATA OBJECTS

▶ Notations and definitions
  • Programming languages
  • Interpreters
  • Compilers
  • Specialisers

▶ Time costs:
  • Interpretation overhead
  • Compiling versus interpretation
  • Effect of double interpretation
  • A dream: automatically generate compilers from interpreters

▶ Self-interpretation of a subset of \textsc{scheme}

▶ Partial evaluation systems
SINGLE-LEVEL INTERPRETATION COSTS:

- Let $p$ be an $S$-program,
- Let $time^S_p(d)$ be the time to run $p$ on input $d$ if one had an $S$-processor and
- let $time^L_{int}(p, d)$ be the time to interpret $p$ on an $L$-processor.
- Typically:

  $$5 \cdot time^S_p(d) \leq time^L_{int}(p, d) \leq 100 \cdot time^S_p(d)$$

Interpretation is expensive!
WHY IS INTERPRETATION SLOW?

It takes time to

1. perform elementary operations specified in source program $p$
2. do the “interpretation loop”, e.g., syntax analysis and dispatching
3. update interpreter data structures and to
4. access interpreter data structures (e.g., look up a variable’s value or find called procedure)
5. One $S$ operation may require many $L$ operations to simulate

► The overhead is relatively less if the time for elementary operations dominates the interpretation loop, e.g., APL

► double or multiple interpretation levels can make unacceptably large (e.g., $100 \times 100 = 10000$).
Interpretation overhead.

Overhead of multiple levels of interpretation.

Timing function: Extend language definition with time.

Definition A language $L$ consists of

- $L$-programs (a set)
- $L$-data (a set)
- A semantic function

$$\llbracket \bullet \rrbracket^L : L\text{-}programs \rightarrow (L\text{-}data \rightarrow L\text{-}data)$$

plus

- A timing function

$$\text{time}^L : L\text{-}programs \rightarrow (L\text{-}data \rightarrow \mathcal{N})$$

Intuition: $\text{time}^L_p = \text{number of instructions executed when applying program } p \text{ to data } d.$
**INTERPRETATION OVERHEAD**

**Interpretation equation:**

\[
[s_\text{source}]^S(d) = [\text{int}]^L(\text{source}, d)
\]

**Compare:**

- **Direct execution cost:** \(time^S_{\text{source}}(d):\)
- **Interpretation cost:** \(time^L_{\text{int}}(\text{source}, d):\)

**A COMMON RELATION:** for all \(d\)

\[
\alpha \cdot time^S_{\text{source}}(d) \leq time^L_{\text{int}}(\text{source}, d)
\]

where \(\alpha\) is a reasonably large constant
INTERPRETATION VERSUS COMPILING

Interpretation:
\[[\text{source}]^S(d) = [\text{int}]^L(\text{source}, d)\]

- Program and input are provided together at run time.
- Instruction decoding and execution are interleaved at run time.

Compilation:
\[[\text{source}]^S(d) = [[[\text{comp}]^L(\text{source})]]^T(d)\]

This is staged execution:
- Program is provided in advance.
- Decoding performed in advance.
- Execution performed at run time.
INTERPRETATION COMBINED WITH COMPILING

Interpretation:

\[
[s_{source}]^{S}(d) = [\text{int}]^{L}(source, d)
\]

- Program and input are provided together at run time.
- Instruction decoding and execution are interleaved at run time.

Compilation:

\[
[s_{source}]^{S}(d) = [[[\text{comp}]^{L}(source)]^{T}}(d)
\]

Stage 1: Program provided, instruction decoding. Stage 2: Execution.

Combination:

\[
[s_{source}]^{S}(d) = [\text{int}]^{L}([\text{comp}](source), d)
\]

Stage 1: Compile into a more easily or efficiently interpreted form. Stage 2: Execution.

Stage 1 may include: Flow analysis, abstract interpretation, type checking, other static analyses and transformations
SOURCES OF INTERPRETIVE OVERHEAD

Time to process instructions:
- Identify the instruction to execute, i.e., “dispatch on syntax”
- Then perform the action it specifies

Overhead: the time to identify instructions.

Cost $c$ depends on language size. Often around 10 for simple interpreters.

Time to reference variables:
- Look variable location up in symbol table
- Reference associated location

Overhead: time to look up variables.

Cost $f(p)$ depends on program size.

Add these to get the overhead per instruction:

$$\alpha = c + f(p)$$
MULTIPLE LEVELS OF INTERPRETATION

Let $L_2 = \text{PROLOG}$, $L_1 = \text{LISP}$, $L_0 = \text{LISP}$, RISC machine code. If Prolog programs are processed interpretively by an interpreter written in LISP; and the LISP code is processed by an interpreter written in RISC code.

Two interpretation levels

- interpreter $\text{int}_1^2$ written in L1 implements L2; and
- interpreter $\text{int}_0^1$ written in L0 implements L1.

By definition of an interpreter (applied twice)

$$[[p_2]]^{L_2}(d) = [[\text{int}_1^2]]^{L_1}(p_2,d) = [[\text{int}_0^1]]^{L_0}(\text{int}_1^2,(p_2,d))$$
For appropriate constants $\alpha_{01}$, $\beta$ and any $L_1$-program $p_1$, $L_2$-program $p_2$ and data $d$,

**Timing equations:**

\[
\alpha \cdot \text{time}_{L_1}^{p_1}(d) \leq \text{time}_{\text{int}_0}^{L_0}(p_1, d) \quad \text{and} \\
\beta \cdot \text{time}_{L_2}^{p_2}(d) \leq \text{time}_{\text{int}_1}^{L_1}(p_2, d)
\]

**Cost of two levels of interpretation:** A little algebra gives

\[
\alpha \cdot \beta \cdot \text{time}_{p_2}^{L_2}(d) \leq \text{time}_{\text{int}_0}^{L_0}(\text{int}_1^2, (p_2, d))
\]

Note the **multiplication** of interpretive overheads.
The cost of one interpretation level may be acceptable price to pay, to have a powerful, expressive language (LISP since its beginnings).

BUT for several levels, each new interpreter multiplies time by a significant constant factor.
Total interpretive overhead may be excessive (and often is).

Compilation is clearly preferable to having several interpreters, each interpreting the next.
OUTLINE: PROGRAMS AS DATA OBJECTS

▶ Notations and definitions
  • Programming languages
  • Interpreters
  • Compilers
  • Specialisers

▶ Time costs:
  • Interpretation overhead
  • Compiling versus interpretation
  • Effect of double interpretation
  • A dream: automatically generate compilers from interpreters

▶ Self-interpretation of a subset of SCHEME

▶ Partial evaluation systems
We will see how to use partial evaluation to convert

- an interpreter for $S$ written in $L$ into
- a compiler from $S$ into $L$

\[ S \quad \rightarrow \quad S \rightarrow L \]

This is 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 (Reynolds!).

- Compiler correctness: the question is avoided, since the compiler will always be faithful to the interpreter from which it was generated.
OUTLINE: PROGRAMS AS DATA OBJECTS

▶ Notations and definitions
  • Programming languages
  • Interpreters
  • Compilers
  • Specialisers

▶ Time costs:
  • Interpretation overhead
  • Compiling versus interpretation
  • Effect of double interpretation
  • A dream: automatically generate compilers from interpreters

▶ Self-interpretation of a subset of SCHEME

▶ Partial evaluation systems
An interpreter that can interpret its own implementation.

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

(Non) example:

Language: \( e ::= n \\
| (+ e_1 e_2) \\
(\text{DEFINE} (\text{interp} e) \\
(\text{if} \\
((\text{number?} e) e) \\
((\text{equal?} (\text{head} e) "+") \\
(+ (\text{interp} (\text{head}(\text{tail} e))) (\text{interp} (\text{head}(\text{tail}(\text{tail} e)))))) \\
(\text{else} (\text{error} "\text{illegal expression}")))) \\
\text{#<unspecified> \\
> (\text{interp} (\text{define} (\text{interp} e) (\text{if} \ldots)))) \\
\text{ERROR: illegal expression} \]
WHY A SELF-INTERPRETER?

\[ [\text{int}]^L(\text{source}, d) = [\text{int}]^L(\text{int}, (\text{source}, d)) \]

If we can run the language directly, why interpret it?

▶ Curiosity  
(does self-application really work?)

▶ Theory counterexamples  
(e.g., unsolvability of the halting problem)

▶ Self-processing tools used in practice:
  
  ● Compiler: compiler that runs faster.  
  (bootstrapping!)
  
  ● Specialiser: next lecture.  
  (Futamura projections)
  
  ● Both analogous to a self-interpreter.
Details given in notes, Section 3.4

Can be downloaded and run. Find at DIKU, at

/usr/local/del1/datV-progsprog/ExerciseFiles

Can run itself, and run itself running itself, ... See Exercise 3.5.
Notations and definitions

- Programming languages
- Interpreters
- Compilers
- Specialisers

Time costs:

- Interpretation overhead
- Compiling versus interpretation
- Effect of double interpretation
- A dream: automatically generate compilers from interpreters

Self-interpretation of a subset of SCHEME

Partial evaluation systems
SPECIALISATION EXAMPLE 1: POWER

A two-input program

\[ p = \]

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

Program \( p \), specialised to static input \( n = 5 \):

\[ p_5 = \]

\[ f_5(x) = x \times (x \times (x \times (x \times (x \times 1)))) \]

Or, with a better algorithm:

A two-input program

\[ p = \]

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

Program \( p \), specialised to static input \( n = 5 \):

\[ p_5 = \]

\[ f_5(x) = x \times ((x^2)^2) \]
EXAMPLE 2: ACKERMANN’S FUNCTION

Specialisation of a program to compute Ackermann’s function (known from mathematical logic):

\[
a(m,n) = \begin{cases} 
  n+1 & \text{if } m = 0 \\
  a(m-1,1) & \text{if } n = 0 \\
  a(m-1,a(m,n-1)) & \text{otherwise}
\end{cases}
\]

Suppose we specialise to known \( m = 2 \) (but unknown \( n \)).
Partial evaluation can yield \( p_2 \), about twice as fast:

\[
a_2(n) = \begin{cases} 
  3 & \text{if } n = 0 \\
  a_1(a_2(n-1)) & \text{otherwise}
\end{cases}
\]

\[
a_1(n) = \begin{cases} 
  2 & \text{if } n = 0 \\
  a_1(n-1) + 1 & \text{otherwise}
\end{cases}
\]

Techniques used:
► symbolic computation
► unfolding function calls,
► function name specialisation

In this example function name “a” gets specialised into \( a_1 \) and \( a_2 \).
For the **SCHEME** language:

- **Similix** (Copenhagen, Bondorf, ...)
  This can be downloaded from DIKU at
  
  /usr/local/topps/mix/lib/unmix/

- **For learning**: **UNMIX** (Moscow, Romanenko, ...). Can be downloaded from the same place

- **PGG** (Freiburg, Thiemann, ...)

- **Schism** (Yale/Bordeaux, Consel, ...)

For the **C** language: **Tempo** (Bordeaux, Consel, ...) and **C-Mix** (Copenhagen, Makholm, ...)

For **Prolog**: several systems (Düsseldorf, ...)

Several others
Interpreters, etc. all \textit{L-programs}

and \(d, s \in S\text{-data}\)

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

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

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

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

\textit{spec} \textbf{is a specialiser (from} \(S\text{ to} \text{ T written in} \text{ L}) \textit{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, $time_{ps}(d) < time_p(s, d)$, so there is an efficiency payoff when $ps$ is run on many different $d$’s.

▶ Sometimes, even $time_{spec}(p, s) + time_{ps}(d) < time_p(s, d)$, so there can be an efficiency payoff even for one run of $ps$. 
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:

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

5. Similarly:

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

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

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

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

\[ a_2(n) = \text{if } n =\neq 0 \text{ then } 3 \text{ else } a_1(a_2(n-1)) \]
\[ a_1(n) = \text{if } n =\neq 0 \text{ then } 2 \text{ else } a_1(n-1)+1 \]
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 target, compiler and cogen.

1. \( \text{target} := \lbrack\text{spec}\rbrack(\text{int}, \text{source}) \)
2. \( \text{compiler} := \lbrack\text{spec}\rbrack(\text{spec}, \text{int}) \)
3. \( \text{cogen} := \lbrack\text{spec}\rbrack(\text{spec}, \text{spec}) \)

(We will see the motivation behind these definitions shortly.)
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.
Recall functionality of interpreter \texttt{int}:

$$\text{out} = \texttt{[int]}(\texttt{source, in})$$

Suppose we want to compute

\begin{align*}
\text{out}_1 &= \texttt{[int]}(\texttt{source, in}_1) \\
\text{out}_2 &= \texttt{[int]}(\texttt{source, in}_2) \\
&\ldots
\end{align*}

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

$$\text{target} := \texttt{[spec]}(\texttt{int, source}) (= \texttt{int}_{\texttt{source}})$$

Then

$$\texttt{[target]}(\texttt{in}_i) = \texttt{[[spec]}(\texttt{int, source})\texttt{]}(\texttt{in}_i)$$

$$= \texttt{[int]}(\texttt{source, in}_i)$$

$$= \text{out}_i$$

So \texttt{target} \textbf{is indeed a correct target program wrt. int and source.}
MORE ON FUTAMURA PROJECTION 1: A PARTIAL EVALUATOR CAN COMPILE

Output program $\text{target} = \llbracket \text{spec} \rrbracket (\text{int}, \text{source})$ will be a correctly compiled version of input program $\text{source}$ if

$$\llbracket \text{source} \rrbracket^S = \llbracket \text{target} \rrbracket (= \llbracket \text{target} \rrbracket^L)$$

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

- $\text{out} = \llbracket \text{source} \rrbracket^S (\text{in})$ Assumption
- $\quad = \llbracket \text{int} \rrbracket (\text{source}, \text{in})$ Interpreter def’n
- $\quad = \llbracket \llbracket \text{spec} \rrbracket (\text{int}, \text{source}) \rrbracket (\text{in})$ Specialiser def’n
- $\quad = \llbracket \text{target} \rrbracket (\text{in})$ target def’n

Thus program $\text{target}$ deserves its name. Assuming the partial evaluator is correct, this always yields

$\text{target programs that are correct}$

$\text{with respect to the source programs}$
Recall definition of target program \texttt{target}:
\[
\text{target} = [[\text{spec}]](\text{int}, \text{source})
\]

Suppose we want to compute
\[
\begin{align*}
\text{target}_1 &= [[\text{spec}]](\text{int}, \text{source}_1) \\
\text{target}_2 &= [[\text{spec}]](\text{int}, \text{source}_2) \\
& \vdots
\end{align*}
\]
(That is, compile several different source programs.)

Idea: take
\[
\text{compiler} := [[\text{spec}]](\text{spec}, \text{int}) (= \text{spec}_{\text{int}})
\]
Then

\[[\text{compiler}](\text{source}_i) = [[[\text{spec}](\text{spec, int})](\text{source}_i) = [[\text{spec}](\text{int, source}_i) = \text{target}_i\]

So **compiler is indeed a correct compiler wrt. int.**

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

Hard to understand operationally — but gives good results in practice.
Recall definition of compiler:

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

Suppose we want to compute

\begin{align*}
\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
\end{align*}

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

Idea: take

$$
c\text{ogen} := \llbracket \text{spec} \rrbracket (\text{spec}, \text{spec}) \ (\equiv \text{spec}_{\text{spec}})
$$
CLAIM: \([[\text{spec}]](\text{spec, spec})\) IS A COMPILER

Then

\[
[[\text{cogen}}](\text{int}_i) = [[[\text{spec}]](\text{spec, spec})](\text{int}_i) \\
= [[\text{spec}]](\text{spec, int}_i) \\
= \text{compiler}_i
\]

So \text{cogen} is indeed a correct compiler generator.

\text{cogen} is constructed by \textbf{double self-application} — using \text{spec} to specialise itself to itself.

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

(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 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 = $[[\text{spec}]] (\text{Norma-int}, \text{source})$:

```
(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 $2 \times x + 2$; **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.