Partial Evaluation:

Types, Binding Times and Optimal Specialisation

Lecture 3: The Types Involved in Partial evaluation

Neil D. Jones
DIKU, University of Copenhagen (prof. emeritus)
I: QUICK REVIEW OF 1. ORDER PARTIAL EVALUATION

Programs are data objects in a first order data domain $D$, for example

$$D = \text{Atom} \cup D \times D$$

A programming language $L$ is a set $L$-programs together with a semantic function

$$[\_]^L : L\text{-programs} \rightarrow D \rightarrow D$$

Program meanings are partial functions:

$$[[p]]^L : D \rightarrow D$$

(Omit $L$ if clear from context.) Examples for $L = \text{Lisp}$:

$$[[\text{quote ALPHA}]]^L = \text{ALPHA}$$

$$[[\text{lambda (x) (+ x x)}]]^L 3 = 6$$
An **interpreter** `int` (for `S` written in `L`) must satisfy:

\[
\llbracket \text{source} \rrbracket_S(d) = \llbracket \text{int} \rrbracket(\text{source}.d)
\]

A **compiler** `comp` (from `S` to `T`, written in `L`)

\[
\llbracket \text{source} \rrbracket_S(d) = \llbracket\llbracket \text{comp} \rrbracket(\text{source}) \rrbracket_T(d)
\]

A **partial evaluator** (for `L`) is a program `spec` satisfying, for any program `p` and data `s, d`:

\[
\llbracket p \rrbracket(s.d) = \llbracket\llbracket \text{spec} \rrbracket(p.s) \rrbracket(d)
\]
Applying base functions to known data

unfolding function calls

creating one or more **specialised program points**

Example. Ackermann’s function with known $n = 2$:

$$a(m,n) = \begin{cases} n+1 & \text{if } m=0 \text{ then} \\ a(m-1,1) & \text{if } n=0 \\ a(m-1,a(m,n-1)) & \text{else} \end{cases}$$

**Specialised program:**

$$a2(n) = \begin{cases} 3 & \text{if } n=0 \\ a1(a2(n-1)) & \text{else} \end{cases}$$

$$a1(n) = \begin{cases} 2 & \text{if } n=0 \\ a1(n-1)+1 & \text{else} \end{cases}$$

Less than half as many arithmetic operations as the original: since all tests on and computations involving $m$ have been removed.
1. A partial evaluator can compile:

\[
\text{target} \; \overset{def}{=} \; [[\text{spec}]](\text{int.source})
\]

2. A partial evaluator can generate a compiler:

\[
\text{comp} \; \overset{def}{=} \; [[\text{spec}]](\text{spec.int})
\]

3. A partial evaluator can generate a compiler generator:

\[
\text{cogen} \; \overset{def}{=} \; [[\text{spec}]](\text{spec.spec})
\]
Simple equational reasoning to verify:

1. \([\text{target}] (d) \triangleq [\text{source}]_{S}(d)\)
2. \(\text{target} \triangleq [\text{comp}](\text{source})\)
3. \(\text{comp} \triangleq [\text{cogen}](\text{int})\)

(Surprise! It works well on the computer too... )

Practice: tricky it took a year to get right the first time, in 1984.)
Isn’t there a type error somewhere?

Self-application \( f(f) \) requires \( f \)-type \( A = A \rightarrow A \) (?)

A symbolic version of an operation on values is a corresponding operation on program texts.

The symbolic composition of programs \( p, q \) yields an output program \( r \).

The meaning of \( r = \) the composition of the meanings of \( p \) and \( q \).

Partial evaluation = the symbolic specialisation of a function to a known first argument value.
A notation for the types of symbolic operations. The notation distinguishes

- the types of values from
- the types of program texts

Natural definitions of type correctness of a first-order interpreter, compiler or partial evaluator.

State the problem of optimal partial evaluation.

Show why it’s difficult for typed languages (even first-order).

Reference a solution by Henning Makholm.
UNDERBAR TYPES FOR SYMBOLIC COMPUTATION

\[ t: \text{type} ::= \text{firstorder} \mid \text{type} \times \text{type} \mid \text{type} \rightarrow \text{type} \]

\mid \text{type}_X

Type firstorder describes values in \( D \).

For each language \( X \) and type \( t \), a type constructor

\[ t_X \]

Meaning: the set of \( X \)-programs that denote values of type \( t \).

Examples

- Atom ALPHA has type firstorder
- Lisp program (quote ALPHA) has type \text{firstorder} Lisp
THE MEANING OF TYPE EXPRESSION $T$ IS $[[T]]$

\[
[[\text{firstorder}]] = D
\]

\[
[[t_1 \rightarrow t_2]] = [[[t_1]] \rightarrow [[[t_2]]]]
\]

\[
[[t_1 \times t_2]] = \{(t_1, t_2) \mid t_1 \in [[[t_1]]], t_2 \in [[[t_2]]]\}
\]

\[
[[t_X]] = \{ p \in D \mid [p]^X \in [[[t]]]\}
\]

Some type inference rules:

\[
\frac{\text{exp}_1 : t_2 \rightarrow t_1, \text{exp}_2 : t_2}{\text{exp}_1 \text{exp}_2 : t_1}
\]

\[
\frac{\text{exp} : t_X^-}{[[\text{exp}]]^X : t}
\]

\[
\text{firstordervalue} : \text{firstorder}
\]

\[
\text{exp} : \text{firstorder}
\]
(α → β) = the set of all programs that compute a function from α to β.
Point: no intermediate symbol b is ever produced.
Consider composition \(\text{oneto} ; \text{squares} ; \text{sum}\) where

\[
\begin{align*}
\text{oneto}(n) &= [n, n-1, \ldots, 2, 1] \\
\text{squares}[a_1, a_2, \ldots a_n] &= [a_1^2, a_2^2, \ldots, a_n^2] \\
\text{sum}[a_1, a_2, \ldots a_n] &= a_1 + a_2 + \ldots + a_n.
\end{align*}
\]

Straightforward program:

\[
\begin{align*}
f(n) &= \text{sum}(\text{squares}(\text{oneto}(n))) \\
\text{squares}(l) &= \text{if } l = [] \text{ then } [] \text{ else} \\
&\quad \text{cons}(\text{head}(l)**2, \text{squares}(\text{tail}(l))) \\
\text{sum}(l) &= \text{if } l = [] \text{ then } 0 \text{ else} \\
&\quad \text{head}(l) + \text{sum}(\text{tail}(l)) \\
\text{oneto}(n) &= \text{if } n = 0 \text{ then } [] \text{ else } \text{cons}(n, \text{oneto}(n-1))
\end{align*}
\]

Result of “deforestation”:

\[
g(n) = \text{if } n = 0 \text{ then } 0 \text{ else } n**2+g(n-1)
\]
PARTIAL EVALUATION

\[(\alpha \times \beta \to \gamma) \times \alpha\] \quad \text{pgm-spec} \quad \rightarrow \quad \[(\beta \to \gamma)\]

\[\begin{bmatrix} - \end{bmatrix} \times \text{Identity} \]

\[(\alpha \times \beta \to \gamma) \times \alpha\] \quad \text{fcn-spec} \quad \rightarrow \quad \[(\beta \to \gamma)\]
A BETTER DEFINITION OF PARTIAL EVALUATION

Type in the diagram:
\[ \text{pgm} \rightarrow \text{spec} : (\alpha \times \beta \rightarrow \gamma \times \alpha) \rightarrow (\beta \rightarrow \gamma) \]

First Curry:
\[ \alpha \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma \]

Then generalize:
\[ \text{spec} : \forall \alpha . \forall \tau . \alpha \rightarrow \tau \rightarrow \alpha \rightarrow \tau \]

Usually \( \alpha \) must be first order.

Definition. Program \( \text{spec} \in D \) is a partial evaluator if for all \( p, s \in D \),

\[ [[p]] s \approx [[[\text{spec}]] p s] \]
An **interpreter** `int` (for `S` written in `L`) must satisfy:

\[[\text{source}]_S \cdot \triangleq [\text{int}]\text{source}\]

A **compiler** `comp` (from `S` to `T`, written in `L`)

\[[\text{source}]_S \cdot \triangleq [[[\text{comp}]\text{source}]_T]\]

A **partial evaluator** (for `L`) is a program `spec` satisfying, for any program `p` and data `s`:

\[[p]_s \cdot \triangleq [[[\text{spec}]_p s]]\]
The Futamura projections:

\[
\begin{align*}
[\text{spec}] \text{int source} & \overset{\text{def}}{=} \text{target} \\
[\text{spec}] \text{spec int} & \overset{\text{def}}{=} \text{compiler} \\
[\text{spec}] \text{spec spec} & \overset{\text{def}}{=} \text{cogen}
\end{align*}
\]

Do these type-check?

Recall our type inference rules:

\[
\begin{align*}
\text{exp}_1 : t_2 \rightarrow t_1, \quad \text{exp}_2 : t_2 & \quad \Rightarrow \quad \text{exp}_1 \text{exp}_2 : t_1 \\
\text{exp} : t & \quad \Rightarrow \quad [\text{exp}]^X : t \\
\text{exp} : t & \quad \Rightarrow \quad \text{exp} : \text{firstorder}
\end{align*}
\]
1. Type of source: $\tau_S$

2. Type of [int]: $\forall \tau. \tau_S \rightarrow \tau$

3. Type of [compiler]: $\forall \tau. \tau_S \rightarrow \tau_T$

4. Type of [spec]: $\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta$
   where $\alpha$ is first order

Remark: Line 3 gives
• the type of the compiling function.
• The type of the compiler text is:

   compiler: $\forall \tau. \tau_S \rightarrow \tau_T$
We wish to find the type of

$$\text{target} \overset{\text{def}}{=} \left[ \text{spec} \right] \text{int source}$$

Assume program source has type $\tau_S$. A deduction:

$$\left[ \text{spec} \right] : \rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma \quad \left[ \text{spec} \right] : \tau_S \rightarrow \tau \rightarrow \tau_S \rightarrow \tau \quad \text{int} : \tau_S \rightarrow \tau \quad \text{source} : \tau_S$$

$$\left[ \text{spec} \right] \text{int source} : \tau_s$$
Thus \texttt{target} has type

\[ \tau = \tau_L \]

(as expected).

The deduction uses only the type inference rules and generalization of polymorphic variables.
Recall that: compiler \( \overset{def}{=} [\text{spec}] \text{spec int where} \)

\[
\text{interpreter int has type } \forall \tau . \tau \mathsf{S} \to \tau .
\]

We show: If

\[
p \text{ has type } \alpha \to \beta
\]

then

\[
[\text{spec}] \text{spec } p \text{ has type } \alpha \to \beta
\]

Deduction:

\[
\text{spec} : \alpha \to \beta \to \alpha \to \beta \\
[\text{spec}] : \alpha \to \beta \to \alpha \to \beta
\]

\[
[p] : \alpha \to \beta
\]

\[
[\text{spec}] \text{spec } p : \alpha \to \beta
\]
Recall that:

\[ \text{compiler} \overset{\text{def}}{=} \llbracket \text{spec} \rrbracket \text{spec} \text{int} \]

where \text{interpreter} \text{ int} \text{ has type } \forall \tau \cdot \tau \rightarrow S \rightarrow \tau.

We just showed: If \( p \) \text{ has type } \alpha \rightarrow \beta \text{ then } \llbracket \text{spec} \rrbracket \text{spec } p \text{ has type } \alpha \rightarrow \beta.

Substituting \( \alpha = \tau \rightarrow S \), \( \beta = \tau \), we get

\[ \text{compiler} = \llbracket \text{spec} \rrbracket \text{spec} \text{int} : \tau \rightarrow S \rightarrow \tau \]

and so (as desired)

\[ \llbracket \text{compiler} \rrbracket : \tau \rightarrow S \rightarrow \tau \]
We just showed that:

\[
\llbracket \text{compiler} \rrbracket : \tau \xrightarrow{S} \tau
\]

Furthermore \( \tau \) was arbitrary, so

\[
\llbracket \text{compiler} \rrbracket : \forall \tau. \tau \xrightarrow{S} \tau
\]

By similar reasoning (too big a tree to show!):

\[
\llbracket \text{cogen} \rrbracket : \forall \alpha \forall \beta. \alpha \xrightarrow{\beta} \alpha \xrightarrow{\beta}
\]
Suppose \( \text{sint} \) is a self-interpreter and \( p, p' \) are programs such that

\[
p' = [\text{spec}] \text{sint} p
\]

**Correctness of** \( \text{spec} \) **implies**

\[
[p'] = [p]
\]

**but** \( p, p' \) **need not be the same programs.**
Definition Partial evaluator \( \text{spec} \) is optimal if it removes all interpretational overhead: For a natural self-interpreter \( \text{sint} \) and any program \( p \) and input \( d \),

\[
\text{time}_{p'}(d) \leq \text{time}_p(d)
\]

Intuitively: \( \text{spec} \) has removed an entire layer of interpretation.
Example. Ackermann’s function with known $n = 2$:

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

Specialised program:

$$a_2(n) = \begin{cases} 3 & \text{if } n=0 \\ a_1(a_2(n-1)) & \text{else} \end{cases}$$

$$a_1(n) = \begin{cases} 2 & \text{if } n=0 \\ a_1(n-1)+1 & \text{else} \end{cases}$$

where $a_1(n) = a(1,n)$ and $a_2(n) = a(2,n)$ are specialised versions of function $a$. 
TECHNIQUES FOR PARTIAL EVALUATION

- Applying base functions to known data
- Unfolding function calls
- Creating one or more specialised functions

Specialised Ackermann’s function performs less than half as many arithmetic operations as the original:

All computations involving \( m \) have been removed.
A well-known trick: split the environment into two parallel lists:

\[ ns = (n_1, \ldots, n_k) \]
\[ vs = (v_1, \ldots, v_k) \]

Part of the interpreter text:

\[
\text{eval(exp,ns,vs,pgm)} = \text{case exp of}
\]
"X" : lookup X ns vs
"e1+e2" : eval(e1,ns,vs,pgm) + eval(e2,ns,vs,pgm)
...

--- 28 ---
Binding times: exp, ns, pgm are static, while vs is dynamic.

Consequence: all functions in \( p' = \llbracket \text{spec} \rrbracket \text{sint} \ p \) have form:

\[
eval_{\text{exp,ns,pgm}}(vs) = \ldots
\]

An annoying problem:
there is only only one argument in each \( p' \) function (!)

This cannot be optimal, i.e., as fast as \( p \)!
This problem: specialised program $p' = [\text{spec}] \; \text{sint} \; p$ inherits a limit from sint: a specialised function

$$f_{a,b}(x, y) = \ldots$$

has $k' \leq k$ arguments, if sint function $f$ has $k$ arguments. Thus no function in $p'$ has more than $k$ arguments(!)

For interpreter function $\text{eval}$, this problem can be solved by variable splitting, also called arity raising.

Observation: for a fixed $p$, the interpreter’s variable $\text{vs}$ always has a constant length $k$. 
Split $\text{eval}_{\exp, \ns, \pgm}(\vs) = \ldots$ into

$$\text{eval}_{\exp, \ns, \pgm}(v_1, \ldots, v_k) = \ldots$$

By this and similar tricks, a first-order “optimal” $\text{spec}$ can be built.

For the “optimal” $\text{spec}$, $p' = [[\text{spec}]] \text{ sint } p$ is

**identical to** $p$, up to the naming of variables (and thereby just as fast).
Interpreter example with types (first-order):

\[
\text{eval} : \quad \text{Exp} \to \text{Names} \to \text{Values} \to \text{Univ}
\]

\[
\text{Univ} = \text{Int integer} \mid \text{Pair Univ} \times \text{Univ} \mid \ldots
\]

\[
\text{eval exp ns vs} = \text{case exp of}
\]

- "X" : env X
- "e1:e2" : Pair (eval e1 ns vs) (eval e2 ns vs)
  
  ...

Suppose source program has type \( [[p]] : \mathcal{N} \to \mathcal{N} \).

Then specialised program has a different type:

\[
[[p']] : \text{Univ} \to \text{Univ}
\]

Significantly less efficient. With higher-order types: even worse!
To achieve optimal specialisation for a typed programming language.

- Stated in 1987
- Unsuccessfully attempted for a number of years
- Solved by Henning Makholm in 1999. Reported in SAIG 2000 (ICFP workshop at Montreal)
Type of a specialiser:

\[
[[\text{spec}]] : \text{Pgm} \rightarrow \text{Data} \rightarrow \text{Pgm}
\]

This is correct, but it “doesn’t tell the whole story”

To clarify the problem, extend \( \alpha \rightarrow \beta \) to

One version for \( L \)-programs:

\[ \frac{\alpha \rightarrow \beta}{\text{Pgm}} \]

and another version for data types:

\[ \frac{\alpha}{\text{Data}} \]

\( \frac{\alpha}{\text{Data}} \) is a subtype of \( \text{Data} \):

the set of encodings of all values of type \( \alpha \)
The type of a specialiser’s meaning, redone:

\[
[[\text{spec}]] : \frac{\alpha \rightarrow \beta \rightarrow \gamma}{Pgm} \rightarrow \frac{\alpha}{Data} \rightarrow \frac{\beta \rightarrow \gamma}{Pgm}
\]

Type of a self-interpreter’s meaning:

\[
\forall \alpha, \beta . [[\text{sint}]] : \frac{\alpha \rightarrow \beta}{Pgm} \rightarrow \frac{\alpha}{Univ} \rightarrow \frac{\beta}{Univ}
\]

and thus

\[
\forall \alpha, \beta . \text{sint} : \frac{\alpha \rightarrow \beta}{Pgm} \rightarrow \frac{\alpha}{Univ} \rightarrow \frac{\beta}{Univ}
\]

Here \( Univ \) is a universal data type.
The optimality criterion: \( p' = \text{[spec]} \text{sint} \ p \) should be as good as \( p \).

Alas this is impossible since:

\[
[p] : \alpha \rightarrow \beta
\]

but

\[
[p'] = [[\text{[spec]} \text{sint} \ p]] : \frac{\alpha}{\text{Univ}} \rightarrow \frac{\beta}{\text{Univ}}
\]
A way out: use a self-interpreter with type

\[
[sint_{\alpha \to \beta}] : \frac{\alpha \to \beta}{Pgm} \to \alpha \to \beta
\]

This can be mechanically obtained from

\[
\forall \alpha, \beta . \ [\text{sint}] : \frac{\alpha \to \beta}{Pgm} \to \frac{\alpha}{Univ} \to \frac{\beta}{Univ}
\]

by

\[
[sint_{\alpha \to \beta}] p a = decode_\beta([\text{sint}] p \ encode_\alpha(a))
\]

Optimality reformulated: for any \([p] : \alpha \to \beta\) the program

\[
p' = [\text{spec}] sint_{\alpha \to \beta} p
\]

is at least as fast as \(p\).
1. $L = \text{a first-order call-by-value language with}$

2. \text{types unit, integer and sum and product types.}$

3. \text{The self-interpreter uses a universal type $Univ$.}$

4. \text{The self-interpreter has been proven correct (Morten Welinder’s phd thesis).}$
Phase 1: specialise using unsophisticated techniques. The output program uses a universal type $\text{Univ}$.

Phase 2: Retype output program, using

- Type erasure analysis that uses
- non-standard type inference for
- types that are infinite regular trees.

Phase 3: an Identity elimination phase, e.g., $\eta$-reductions for product and sum types.

Punch line: It works, and even achieves:

$$[[\text{spec}]] \text{sint sint} =_{\alpha} \text{sint}$$
CONCLUSIONS

Contributions:

▶ A notation for the types of symbolic operations. It distinguishes types of values from types of program texts.

▶ Natural definitions of type correctness of an interpreter or compiler.

▶ Makholm: succeeded in solving a long-standing open problem using the underbar type notation (after some refinement)

More to do:

▶ Better mathematical understanding of the underbar types.

▶ How to prove that an interpreter or compiler has the desired type?