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

$$\llbracket - \rrbracket^L : L\text{-programs} \rightarrow D \rightarrow D$$

Program meanings are partial functions:

$$\llbracket p \rrbracket^L : D \rightarrow D$$

(Omit $L$ if clear from context.) Examples for $L =$ Lisp:

$$\llbracket (\text{quote } 37) \rrbracket^L = 37$$

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

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

A **compiler** \( \text{comp} \) (from \( S \) to \( T \), written in \( L \))

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

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

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

unfolding function calls

creating one or more specialised program points

Example. Ackermann’s function with known \( n = 2 \):

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

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:
   \[
   \text{target} \overset{def}{=} \text{[[spec]]}(\text{int}, \text{source})
   \]

2. A partial evaluator can generate a compiler:
   \[
   \text{comp} \overset{def}{=} \text{[[spec]]}(\text{spec}, \text{int})
   \]

3. A partial evaluator can generate a compiler generator:
   \[
   \text{cogen} \overset{def}{=} \text{[[spec]]}(\text{spec}, \text{spec})
   \]
CORRECTNESS OF THE FUTAMURA PROJECTIONS

Simple equational reasoning to verify:

1. \([\text{target}]_S(d) \equiv [\text{source}]_S(d)\)
2. \(\text{target} \equiv [\text{comp}](\text{source})\)
3. \(\text{comp} \equiv [\text{cogen}](\text{int})\)

(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?

**Tension:** it seems that self-application \( f(f) \) requires \( f \)-type

\[
A = A \rightarrow A(??)
\]

**Resolution:** Work with symbolic operations. A symbolic version of an operation on values is a corresponding operation on program texts.

**Symbolic composition** of programs \( p, q \) yields a program \( r \).

**Requirement:** 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.
t: type ::= firstorder | type × type | type → type

| type_X

Type `firstorder` describes values in $D$.

For each language $X$ and type $t$, a type constructor

$$
\frac{t}{X}
$$

Meaning: the set of $X$-programs that denote values of type $t$.

Examples

- Atom `37` has type `firstorder`
- Lisp program `(quote 37)` has type `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:

\[
\begin{align*}
\text{exp}_1 : t_2 & \rightarrow t_1, \quad \text{exp}_2 : t_2 \\
\text{exp}_1 \text{exp}_2 : t_1 \\
\text{firstordervalue} : \text{firstorder}
\end{align*}
\]

\[
\begin{align*}
\text{exp} : t^X \\
[[\text{exp}]]^X : t \\
\text{exp} : t^X \\
\text{exp} : \text{firstorder}
\end{align*}
\]
\[(\alpha \rightarrow \beta) \times (\beta \rightarrow \gamma)\];

\[\text{[-]} \times \text{[-]}\];

\[(\alpha \rightarrow \beta) \times (\beta \rightarrow \gamma)\];

\[(\alpha \rightarrow \gamma)\];

\[(\alpha \rightarrow \beta) = \text{the set of all programs that compute a function from } \alpha \text{ to } \beta.\]
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:

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

Result of “deforestation”: 

\[
g(n) = \begin{cases} 0 & \text{if } n = 0 \text{ then } 0 \text{ else} \\ & n**2+g(n-1) \end{cases}
\]
$$\overline{(\alpha \times \beta \rightarrow \gamma) \times \alpha} \xrightarrow{[-]} ID \xrightarrow{[-]} \overline{(\beta \rightarrow \gamma)}$$
A BETTER DEFINITION OF PARTIAL EVALUATION

Type in the diagram:

$$\text{pgm-spec} : (\alpha \times \beta \rightarrow \gamma \times \alpha) \rightarrow (\beta \rightarrow \gamma)$$

First Curry the specialiser:

$$\alpha \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$$

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 $p, s \in D$,

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

\[
[[\text{source}]]_S \overset{*}{=} [[\text{int}]]_{\text{source}}
\]

A compiler \( \text{comp} \) (from \( S \) to \( T \), written in \( L \))

\[
[[\text{source}]]_S \overset{*}{=} [[[\text{comp}]]_{\text{source}}]_T
\]

A partial evaluator (for \( L \)) is a program \( \text{spec} \) satisfying, for any program \( p \) and data \( s \):

\[
[[p]]_s \overset{*}{=} [[[\text{spec}]]_p s]
\]
The Futamura projections:

\[
[\text{spec}] \text{int } \text{source} \quad \text{def} \quad \text{target} \\
[\text{spec}] \text{spec } \text{int} \quad \text{def} \quad \text{compiler} \\
[\text{spec}] \text{spec } \text{spec} \quad \text{def} \quad \text{cogen}
\]

Do these type-check?

Recall our type inference rules:

\[
\frac{\exp_1 : t_2 \to t_1, \exp_2 : t_2}{\exp_1 \exp_2 : t_1}
\]

\[
\frac{\exp : t \quad \text{firstordervalue} : \text{firstorder}}{\text{firstordervalue} : \text{firstorder}}
\]

\[
\frac{\exp : t \quad \text{firstorder}}{\exp : \text{firstorder}}
\]
1. Type of source: \( \tau S \)

2. Type of \([\text{int}]\): \( \forall \tau . \tau S \rightarrow \tau \)

3. Type of \([\text{compiler}]\): \( \forall \tau . \tau S \rightarrow \tau T \)

4. Type of \([\text{spec}]\): \( \forall \alpha . \forall \beta . \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta \)

where \( \alpha \) is first order

Remark: Line 3 gives

▸ the type of the compiling function.

▸ The type of the compiler text is:

\[
\text{compiler} : \forall \tau . \tau S \rightarrow \tau T
\]
We wish to find the type of

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

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

\[
\begin{align*}
[\text{spec}] : & \rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma \\
\text{int} : & \tau_S \rightarrow \tau \\
\text{source} : & \tau_S \\
\hline
[\text{spec}] \text{int source} : & ?
\end{align*}
\]
We wish to find the type of

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

Assume program \text{source} has type \(\tau_S\). A deduction:

\[
\begin{align*}
\text{[spec]} : & \quad \rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma \\
\text{[spec]} : & \quad \tau_S \rightarrow \tau \rightarrow \tau_S \rightarrow \tau \\
\text{int} : & \quad \tau_S \rightarrow \tau \\
\text{source} : & \quad \tau_S
\end{align*}
\]
We wish to find the type of

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

Assume program source has type \( \tau_S \). A deduction:

\[
\begin{align*}
[\text{spec}] : & \quad \tau_S \rightarrow \tau \\
& \quad \tau_S \rightarrow \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{int} : & \quad \tau_S \rightarrow \tau \\
\text{source} : & \quad \tau_S
\end{align*}
\]

\[
[\text{spec}] \text{ int source} : ?
\]
We wish to find the type of

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

Assume program source has type \( \tau_S \). A deduction:

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

\[
[\text{spec}] \text{ int source} : \tau
\]
Thus target has type

$$\tau = \tau_L$$

(as expected).

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

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

We show: If $p$ has type $\alpha \rightarrow \beta$

then $[\text{spec}] \text{spec p}$ has type $\alpha \rightarrow \beta$

Deduction:

\[
[\text{spec}] : \rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma
\]

\[
[\text{spec}] : \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta \quad \text{spec} : \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta
\]

\[
\therefore [\text{spec}] \text{spec} : \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta
\]

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

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

\( \text{compiler} \stackrel{\text{def}}{=} [[\text{spec}]] \text{spec int} \)

where interpreter \( \text{int} \) has type \( \forall \tau . \tau \overset{S}{\to} \tau \).

We just showed: If \( \text{p} \) has type \( \alpha \to \beta \)

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

Substituting \( \alpha = \tau \overset{S}{\to} \tau \), \( \beta = \tau \), we get

\[
\text{compiler} = [[\text{spec}]] \text{spec int} : \tau \overset{S}{\to} \tau
\]

and so (as desired)

\( [[\text{compiler}]] : \tau \overset{S}{\to} \tau \)
We just showed that:

$$[[\text{compiler}]] : \tau S \rightarrow \tau$$

Furthermore $\tau$ was arbitrary, so

$$[[\text{compiler}]] : \forall \tau . \tau S \rightarrow \tau$$

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

$$[[\text{cogen}]] : \forall \alpha \forall \beta . \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta$$
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 for any program $p$ and input $d$, we have:

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

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:**

$$a2(n) = \begin{cases} 3 & \text{if } n=0 \\ a1(a2(n-1)) & \text{else} \end{cases}$$
$$a1(n) = \begin{cases} 2 & \text{if } n=0 \\ a1(n-1)+1 & \text{else} \end{cases}$$

where $a1(n) = a(1,n)$ and $a2(n) = a(2,n)$ 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:

\[
\text{eval(exp,ns,vs,pgm) =}
\]
\[
\begin{array}{cccc}
S & S & D & S
\end{array}
\quad \text{-- binding times --}
\]

\[
\text{case exp of}
\]
\[
\text{"X" : lookup X ns vs}
\]
\[
\text{"e1+e2" : eval(e1,ns,vs,pgm) + eval(e2,ns,vs,pgm)}
\]
\[
\ldots
\]
Binding times: exp, ns, pgm are static, while vs is dynamic.

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

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

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

This cannot be optimal, i.e., as fast as \( p \)!
INHERITED LIMITS DURING SPECIALISATION

This problem: specialised program \( p' = [[\text{spec}]] \text{sint} p \) inherits a limit from \( \text{sint} \): a specialised function

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

has \( k' \leq k \) arguments, if \( \text{sint} \) function \( f \) has \( k \) arguments.

Thus no function in \( p' \) has more than \( k \) arguments(!)

For interpreter function 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 $\text{eval}_{\text{exp,ns,pgm}}(\text{vs}) = \ldots$ into

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

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

For the “optimal” $\text{spec}$, if $p' = [\text{spec}] \ \text{sint p}$ then $p'$ is identical to $p$, up to the naming of variables (and thereby just as fast).
OPTIMALITY IS HARDER FOR TYPED LANGUAGES!

Interpreter example with types (first-order):

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

\[
\text{Univ} = \text{Int} \ | \ \text{Pair Univ} * \text{Univ} \ | \ ...
\]

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

"X" : \(\text{env} \ X\)

"e1:e2" : \(\text{Pair} \ (\text{eval} \ e1 \ ns \ vs) \ (\text{eval} \ e2 \ ns \ vs)\)

...

Suppose *source program* has type

\[[p] : \mathcal{N} \rightarrow \mathcal{N}\]

Then *specialised program* has a different type:

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

Significantly less efficient. With higher-order types: even worse!
A CHALLENGING PROBLEM

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:

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

This “doesn’t tell the whole story”. A problem is that different programs may require/use different data formats for input/output. A more refined notation:

1. Use almost the same underbar notation \(\alpha \rightarrow \beta\) for program meanings:

   All \(L\)-programs \(p\) such that \(\langle p \rangle \in [\alpha \rightarrow \beta]\):

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

2. For values of \(\alpha\) type encoded in another data type:

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

\(Data\) is the set of encodings of all values of type \(\alpha\).

Encoding is conceptually trivial, but computing \([\_\_]\) is not.
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 \textit{Univ} is a universal data type.
Point:
Any (here: first-order) value can be represented as a $Univ$-value
- without loss of information.
- there exist computable encoding and decoding algorithms.

For example:

```haskell
type Univ = UNIT
  | Integer Int
  | Sum (Univ Univ)
  | Prod (Univ Univ)
```
The optimality criterion: \( p' = [[\text{spec}] \ \text{sint} \ p] \) should be as good as \( p \).

Alas this is impossible since:

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

but

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

$$[[\text{sint}_{\alpha \rightarrow \beta}]] : \frac{\alpha \rightarrow \beta}{Pgm} \rightarrow \alpha \rightarrow \beta$$

This can be mechanically obtained from

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

by

$$[[\text{sint}_{\alpha \rightarrow \beta}]] p a = \text{decode}_\beta([[\text{sint}]] p \ \text{encode}_\alpha(a))$$

Optimality reformulated: for any $$[[p]] : \alpha \rightarrow \beta$$ the program

$$p' = [[\text{spec}]] \ \text{sint}_{\alpha \rightarrow \beta} p$$

is at least as fast as $$p$$. 
OPTIMALITY ACHIEVED

1. \( L \) = a first-order call-by-value language with

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

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

4. 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 $U_{niv}$.

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?