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

$$D = Atom \cup D \times D$$

A programming language L **is a set** $\mathrm{L}\text{-}programs$ **together with a semantic function**

$$[\![\_]\!]^{\mathrm{L}} : \mathrm{L}\text{-}programs \longrightarrow D \rightharpoonup D$$

Program meanings are partial functions:

$$[\![\mathrm{p}]\!]^{\mathrm{L}} : D \rightharpoonup D$$

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

$$[\![\texttt{(quote ALPHA)}]\!]^{\mathrm{L}} = \texttt{ALPHA}$$
$$[\![\texttt{(lambda (x) (+ x x))}]\!]^{\mathrm{L}}\ \mathbf{3} = \texttt{6}$$

INTERPRETER, COMPILER, PARTIAL EVALUATOR

An interpreter `int` **(for** S **written in** L**) must satisfy:**

$$[\![\mathtt{source}]\!]_{\mathrm{S}}(\mathtt{d}) \doteq [\![\mathtt{int}]\!](\mathtt{source.d})$$

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

$$[\![\mathtt{source}]\!]_{\mathrm{S}}(\mathtt{d}) \doteq [\![[\![\mathtt{comp}]\!](\mathtt{source})]\!]_{\mathrm{T}}(\mathtt{d})$$

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

$$[\![\mathtt{p}]\!](\mathtt{s.d}) \doteq [\![[\![\mathtt{spec}]\!](\mathtt{p.s})]\!](\mathtt{d})$$

TECHNIQUES FOR PARTIAL EVALUATION

- **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) = if m=0 then n+1 else
         if n=0 then a(m-1,1)
         else a(m-1,a(m,n-1))
```

Specialised program:

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

Less than half as many arithmetic operations as the original: since all tests on and computations involving `m` **have been removed.**

THE FUTAMURA PROJECTIONS

1. **A partial evaluator can** compile:

$$\texttt{target} \stackrel{def}{=} [\![\texttt{spec}]\!](\texttt{int.source})$$

2. **A partial evaluator can** generate a compiler:

$$\texttt{comp} \stackrel{def}{=} [\![\texttt{spec}]\!](\texttt{spec.int})$$

3. **A partial evaluator can** generate a compiler generator:

$$\texttt{cogen} \stackrel{def}{=} [\![\texttt{spec}]\!](\texttt{spec.spec})$$

CORRECTNESS OF THE FUTAMURA PROJECTIONS

Simple equational reasoning to verify:

1. $[\![\texttt{target}]\!](\texttt{d}) \doteq [\![\texttt{source}]\!]_{\mathrm{S}}(\texttt{d})$
2. $\texttt{target} \doteq [\![\texttt{comp}]\!](\texttt{source})$
3. $\texttt{comp} \doteq [\![\texttt{cogen}]\!](\texttt{int})$

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

Practice: tricky it took a year to get right the first time, in 1984.)

II. UNDERBAR TYPES FOR PARTIAL EVALUATION

Isn't there a type error somewhere?

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

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.

REMAINDER OF THIS TALK

- 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

$$t\colon \mathit{type} \;::=\; \mathit{firstorder} \mid \mathit{type} \times \mathit{type} \mid \mathit{type} \rightarrow \mathit{type}$$

$$\mid \quad \underline{\mathit{type}}_{\mathrm{X}}$$

Type $\mathit{firstorder}$ **describes values in** D**.**

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

$$\underline{t}_{\mathrm{X}}$$

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

Examples

- **Atom** `ALPHA` **has type firstorder**
- **Lisp program** `(quote ALPHA)` **has type** $\underline{\mathit{firstorder}}_{\mathrm{Lisp}}$

THE MEANING OF TYPE EXPRESSION T IS $[\![T]\!]$

$$[\![\mathit{firstorder}]\!] = \boldsymbol{D}$$

$$[\![\boldsymbol{t_1} \rightarrow \boldsymbol{t_2}]\!] = [[\![\boldsymbol{t_1}]\!] \rightarrow [\![\boldsymbol{t_2}]\!]]$$

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

$$[\![\, \underline{\boldsymbol{t}}_{\mathbf{X}} \,]\!] = \{\ \mathtt{p} \in \boldsymbol{D} \quad | \quad [\![\mathtt{p}]\!]^{\mathtt{X}} \in [\![\boldsymbol{t}]\!]\ \}$$

Some type inference rules:

$$\frac{\mathit{exp}_1 : t_2 \rightarrow t_1, \quad \mathit{exp}_2 : t_2}{\mathit{exp}_1\, \mathit{exp}_2 : t_1} \qquad \frac{\mathit{exp} : \underline{\boldsymbol{t}}_{\mathbf{X}}}{[\![\mathit{exp}]\!]^{\mathtt{X}} : t}$$

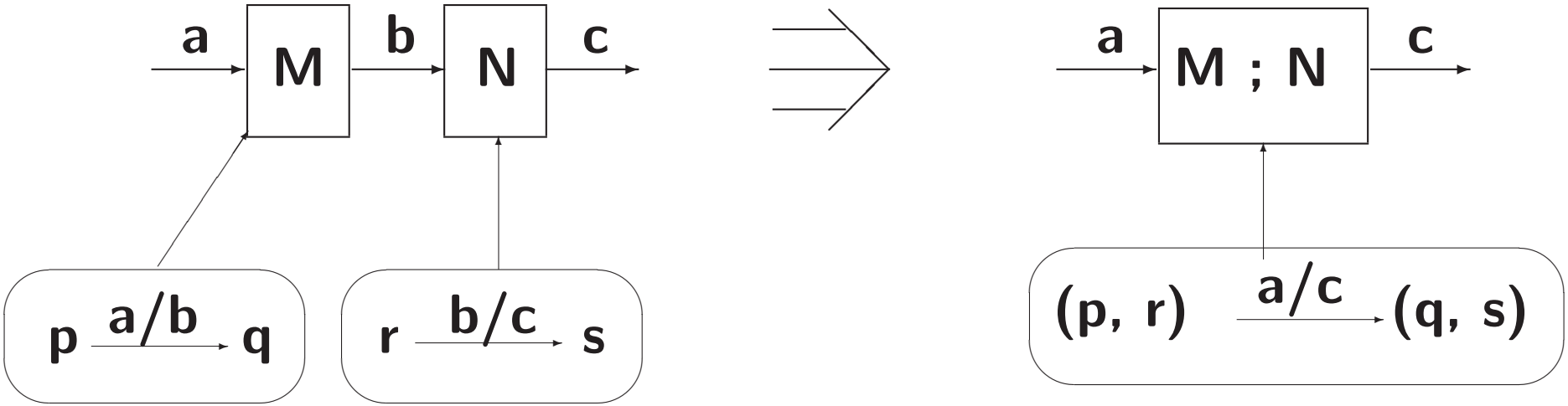
$$\frac{}{\mathit{firstordervalue} : \mathit{firstorder}} \qquad \frac{\mathit{exp} : \underline{\boldsymbol{t}}_{\mathbf{X}}}{\mathit{exp} : \mathit{firstorder}}$$

SYMBOLIC COMPOSITION

$$
\begin{tikzcd}
\boxed{\underline{(\alpha \to \beta)} \times \underline{(\beta \to \gamma)}} \arrow[r, "\underline{;}"] \arrow[d, "{[\![-]\!] \times [\![-]\!]}"'] & \boxed{\underline{(\alpha \to \gamma)}} \arrow[d, "{[\![-]\!]}"] \\
\boxed{(\alpha \to \beta) \times (\beta \to \gamma)} \arrow[r, ";"] & \boxed{(\alpha \to \gamma)}
\end{tikzcd}
$$

$\underline{(\alpha \to \beta)}$ = **the set of all programs that compute a function from α to β.**

COMPOSITION OF FINITE TRANSDUCERS



Point: no intermediate symbol b is ever produced.

COMPOSITION OF PROGRAMS

Consider composition $\boxed{\mathit{oneto}\ ;\ \mathit{squares}\ ;\ \mathit{sum}}$ **where**

$$\begin{aligned}
\mathit{oneto}(\boldsymbol{n}) &= [\boldsymbol{n}, \boldsymbol{n}-\mathbf{1}, \ldots, \mathbf{2}, \mathbf{1}] \\
\mathit{squares}[\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots \boldsymbol{a}_n] &= [\boldsymbol{a}_1^2, \boldsymbol{a}_2^2, \ldots, \boldsymbol{a}_n^2] \\
\mathit{sum}[\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots \boldsymbol{a}_n] &= \boldsymbol{a}_1 + \boldsymbol{a}_2 + \ldots + \boldsymbol{a}_n.
\end{aligned}$$

Straightforward program:

```
f(n)        = sum(squares(oneto(n)))
squares(l)  = if l = [] then [] else
              cons(head(l)**2, squares(tail(l)))
sum(l)      = if l = [] then 0 else
              head(l) + sum(tail(l))
oneto(n)    = if n = 0 then [] else cons(n, oneto(n-1))
```

Result of "deforestation":

```
g(n) = if n = 0 then 0 else n**2+g(n-1)
```

PARTIAL EVALUATION

$$
\begin{tikzcd}
\underline{(\alpha \times \beta \to \gamma)} \times \alpha \arrow[r, "\textbf{pgm-spec}"] \arrow[d, "[\![-]\!] \times \textbf{Identity}"'] & \underline{(\beta \to \gamma)} \arrow[d, "[\![-]\!]"] \\
(\alpha \times \beta \to \gamma) \times \alpha \arrow[r, "\textbf{fcn-spec}"] & (\beta \to \gamma)
\end{tikzcd}
$$

A BETTER DEFINITION OF PARTIAL EVALUATION

Type in the diagram:

$$\mathrm{pgm-spec} : \underline{(\alpha \times \beta \to \gamma} \times \alpha) \to \underline{(\beta \to \gamma)}$$

First Curry:

$$\underline{\alpha \to (\beta \to \gamma)} \to \alpha \to \underline{\beta \to \gamma}$$

Then generalize:

$$\mathrm{spec} : \forall \alpha \,.\, \forall \tau \,.\, \underline{\alpha \to \tau} \to \alpha \to \underline{\tau}$$

Usually α must be first order.

Definition. **Program** $\mathtt{spec} \in D$ **is a partial evaluator if for all** $\mathtt{p}, \ \mathtt{s} \in D$**,**

$$[\![\mathtt{p}]\!]\,\mathtt{s} \doteq [\![[\![\mathtt{spec}]\!]\,\mathtt{p}\,\mathtt{s}]\!]$$

INTERPRETERS, COMPILERS, ETC. REVISITED

An interpreter `int` **(for** S **written in** L**) must satisfy:**

$$[\![\mathtt{source}]\!]_{\mathrm{S}} \doteq [\![\mathtt{int}]\!]\mathtt{source}$$

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

$$[\![\mathtt{source}]\!]_{\mathrm{S}} \doteq [\![[\![\mathtt{comp}]\!]\mathtt{source}]\!]_{\mathrm{T}}$$

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

$$[\![\mathtt{p}]\!]\,\mathtt{s} \doteq [\![[\![\mathtt{spec}]\!]\,\mathtt{p}\,\mathtt{s}]\!]$$

TYPE INFERENCE FOR SELF-APPLICATION

The Futamura projections:

$$[\![\texttt{spec}]\!]\ \texttt{int source} \stackrel{def}{=} \texttt{target}$$

$$[\![\texttt{spec}]\!]\ \texttt{spec int} \stackrel{def}{=} \texttt{compiler}$$

$$[\![\texttt{spec}]\!]\ \texttt{spec spec} \stackrel{def}{=} \texttt{cogen}$$

Do these type-check?

Recall our type inference rules:

$$\frac{exp_1 : t_2 \rightarrow t_1, \quad exp_2 : t_2}{exp_1\, exp_2 : t_1} \qquad \frac{exp : \underline{t}_{\mathbf{X}}}{[\![exp]\!]^{\mathtt{X}} : t}$$

$$\frac{}{firstordervalue : firstorder} \qquad \frac{exp : \underline{t}_{\mathbf{X}}}{exp : firstorder}$$

TYPES OF INTERPRETERS, ETC.

1. Type of `source` : $\underline{\tau}_{\mathrm{S}}$

2. Type of $[\![\texttt{int}]\!]$: $\forall \tau \,.\, \underline{\tau}_{\mathrm{S}} \to \tau$

3. Type of $[\![\texttt{compiler}]\!]$: $\forall \tau \,.\, \underline{\tau}_{\mathrm{S}} \to \underline{\tau}_{\mathrm{T}}$

4. Type of $[\![\texttt{spec}]\!]$: $\forall \alpha \,.\, \forall \beta \,.\, \underline{\alpha \to \beta} \to \alpha \to \underline{\beta}$
   where α is first order

Remark: **Line 3 gives**

- **the type of the compiling function.**
- **The type of the compiler text is:**

$$\texttt{compiler} : \forall \tau \,.\, \underline{\underline{\tau}_{\mathrm{S}} \to \underline{\tau}_{\mathrm{T}}}$$

TYPES DURING COMPILATION

We wish to find the type of

$$\texttt{target} \stackrel{def}{=} [\![\texttt{spec}]\!]\ \texttt{int}\ \texttt{source}$$

Assume program `source` **has type** $\underline{\tau}_{\mathrm{S}}$**. A deduction:**

$$\dfrac{\dfrac{\dfrac{}{[\![\texttt{spec}]\!] : \underline{\rho \to \sigma} \to \rho \to \underline{\sigma}}}{[\![\texttt{spec}]\!] : \underline{\underline{\tau}_{\mathrm{S}} \to \tau} \to \underline{\tau}_{\mathrm{S}} \to \underline{\tau}} \quad \texttt{int} : \underline{\underline{\tau}_{\mathrm{S}} \to \tau}}{[\![\texttt{spec}]\!]\ \texttt{int} : \underline{\tau}_{\mathrm{S}} \to \underline{\tau}} \quad \texttt{source} : \underline{\tau}_{\mathrm{S}}$$

$$\dfrac{[\![\texttt{spec}]\!]\ \texttt{int}\ \texttt{source} : \underline{\tau}}{}$$

TYPES DURING COMPILATION

Thus `target` **has type**

$$\underline{\tau} = \underline{\tau}_{\mathrm{L}}$$

(as expected).

The deduction uses only the type inference rules and generalization of polymorphic variables.

TYPES DURING COMPILER GENERATION: 1

Recall that: $\texttt{compiler} \stackrel{def}{=} [\![\texttt{spec}]\!]\,\texttt{spec}\,\texttt{int}$ **where**

interpreter $\texttt{int}$ **has type** $\forall\tau\,.\,\underline{\tau}_{\mathrm{S}}\!\rightarrow\tau$.

We show: If $\texttt{p}$ **has type** $\underline{\alpha\rightarrow\beta}$

then $[\![\texttt{spec}]\!]\,\texttt{spec}\,\texttt{p}$ **has type** $\underline{\alpha\rightarrow\underline{\beta}}$

Deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\![\texttt{spec}]\!] : \underline{\rho\rightarrow\sigma}\rightarrow\rho\rightarrow\underline{\sigma}}}{[\![\texttt{spec}]\!] : \underline{\underline{\alpha\rightarrow\beta}\rightarrow\alpha\rightarrow\underline{\beta}}\rightarrow\underline{\alpha\rightarrow\beta}\rightarrow\underline{\alpha\rightarrow\underline{\beta}}} \quad \texttt{spec} : \underline{\underline{\alpha\rightarrow\beta}\rightarrow\alpha\rightarrow\underline{\beta}}}{[\![\texttt{spec}]\!]\,\texttt{spec} : \underline{\alpha\rightarrow\beta}\rightarrow\underline{\alpha\rightarrow\underline{\beta}}} \quad \texttt{p} : \underline{\alpha\rightarrow\beta}}{[\![\texttt{spec}]\!]\,\texttt{spec}\,\texttt{p} : \underline{\alpha\rightarrow\underline{\beta}}}$$

TYPES DURING COMPILER GENERATION: 2

Recall that:

$$\texttt{compiler} \stackrel{def}{=} [\![\texttt{spec}]\!]\,\texttt{spec}\,\texttt{int}$$

where interpreter `int` **has type** $\forall\tau\ .\ \underline{\underline{\tau}_{\mathrm{S}} \rightarrow \tau}$.

We just showed: If `p` **has type** $\underline{\alpha \rightarrow \beta}$

then $[\![\texttt{spec}]\!]\,\texttt{spec}\,\texttt{p}$ **has type** $\underline{\alpha \rightarrow \underline{\beta}}$

Substituting $\alpha = \underline{\tau}_{\mathrm{S}}, \beta = \tau$**, we get**

$$\texttt{compiler} = [\![\texttt{spec}]\!]\,\texttt{spec}\,\texttt{int} : \underline{\underline{\tau}_{\mathrm{S}} \rightarrow \underline{\tau}}$$

and so (as desired)

$$[\![\texttt{compiler}]\!] : \underline{\tau}_{\mathrm{S}} \rightarrow \underline{\tau}$$

TYPES DURING COMPILER GENERATION: 3

We just showed that:

$$[\![\texttt{compiler}]\!] : \underline{\tau}_{\mathrm{S}} \rightarrow \underline{\tau}$$

Furthermore τ was arbitrary, so

$$[\![\texttt{compiler}]\!] : \forall \tau \,.\, \underline{\tau}_{\mathrm{S}} \rightarrow \underline{\tau}$$

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

$$[\![\texttt{cogen}]\!] : \forall \alpha \forall \beta \,.\, \underline{\alpha \rightarrow \beta} \rightarrow \underline{\alpha \rightarrow \underline{\beta}}$$

III: OPTIMAL PARTIAL EVALUATION

Suppose $\mathtt{sint}$ **is a self-interpreter and** $\mathtt{p}$, $\mathtt{p}'$ **are programs such that**

$$\mathtt{p}' = [\![\mathtt{spec}]\!]\ \mathtt{sint}\ \mathtt{p}$$

Correctness of $\mathtt{spec}$ **implies**

$$[\![\mathtt{p}']\!] = [\![\mathtt{p}]\!]$$

but $\mathtt{p}$, $\mathtt{p}'$ **need not be the same programs.**

DEFINITION OF OPTIMAL PARTIAL EVALUATION

Definition **Partial evaluator** `spec` **is** optimal **if it removes all interpretational overhead:**
For a natural self-interpreter `sint` **and any program** `p` **and input** `d`**,**

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

Intuitively: `spec` **has removed an entire layer of interpretation.**

PARTIAL EVALUATION EXAMPLE

Example. **Ackermann's function with known** `n = 2`**:**

```
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))
```

Specialised program:

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

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.

EXAMPLE: PART OF A FIRST-ORDER INTERPRETER

A well-known trick: split the environment into two parallel lists:

$$\mathtt{ns} = (\mathtt{n}_1,\ldots,\mathtt{n}_k) \qquad \textbf{names}$$
$$\mathtt{vs} = (\mathtt{v}_1,\ldots,\mathtt{v}_k) \qquad \textbf{values}$$

Part of the interpreter text:

```
eval(exp,ns,vs,pgm) = case exp of
"X"      : lookup X ns vs
"e1+e2"  : eval(e1,ns,vs,pgm) + eval(e2,ns,vs,pgm)
...
```

SPECIALISATION OF THE FIRST-ORDER INTERPRETER

Binding times: `exp,ns,pgm` **are static,**

while `vs` **is dynamic.**

Consequence: all functions in $\mathtt{p}' = [\![\mathtt{spec}]\!]\ \mathtt{sint}\ \mathtt{p}$ **have form:**

$$\mathtt{eval}_{\mathtt{exp},\mathtt{ns},\mathtt{pgm}}(\mathtt{vs}) = \ldots$$

An annoying problem:
there is only only one argument in each $\mathtt{p}'$ **function (!)**

This cannot be optimal, i.e., as fast as $\mathtt{p}$ **!**

INHERITED LIMITS DURING SPECIALISATION

This problem: specialised program $\mathtt{p}' = [\![\mathtt{spec}]\!]\ \mathtt{sint}\ \mathtt{p}$ **inherits a limit from** `sint`**: a specialised function**

$$\mathtt{f}_{\mathtt{a},\mathtt{b}}(\mathtt{x},\mathtt{y}) = \ldots$$

has $k' \leq k$ **arguments, if** `sint` **function** `f` **has** k **arguments. Thus no function in** $\mathtt{p}'$ **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** `vs` **always has a constant length** k**.**

TECHNICAL SOLUTION

Split $\mathtt{eval}_{\mathtt{exp,ns,pgm}}(\mathtt{vs}) = \ldots$ **into**

$$\mathtt{eval}_{\mathtt{exp,ns,pgm}}(\mathtt{v}_1, \ldots, \mathtt{v}_k) = \ldots$$

By this and similar tricks, a first-order "optimal" `spec` **can be built.**

For the "optimal" `spec`, $\mathtt{p}' = [\![\mathtt{spec}]\!]\ \mathtt{sint}\ \mathtt{p}$ **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 :  Exp -> Names -> Values -> Univ
 Univ = Int integer | Pair Univ * Univ | ...
eval exp ns vs = case exp of
"X"      : env X
"e1:e2" : Pair (eval e1 ns vs) (eval e2 ns vs)
...
```

Suppose source program has type $[\![\mathrm{p}]\!] : \mathcal{N} \longrightarrow \mathcal{N}$.

Then specialised program has a different type:

$$[\![\mathrm{p}']\!] : \mathtt{Univ} \longrightarrow \mathtt{Univ}$$

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)**

MAKHOLM'S SOLUTION

Type of a specialiser:

$$[\![\mathtt{spec}]\!] : \mathit{Pgm} \longrightarrow \mathit{Data} \longrightarrow \mathit{Pgm}$$

This is correct, but it "doesn't tell the whole story"

To clarify the problem, extend $\underline{\alpha \longrightarrow \beta}_{\mathrm{L}}$ **to**

One version for L**-programs:** $\dfrac{\alpha \longrightarrow \beta}{\mathit{Pgm}}$

and another version for data types: $\dfrac{\alpha}{\mathit{Data}}$

$\dfrac{\alpha}{\mathit{Data}}$ **is a subtype of** Data**:**

the set of encodings of all values of type α

OPTIMALITY REVISITED

The type of a specialiser's meaning, redone:

$$[\![\texttt{spec}]\!] : \frac{\alpha \to \beta \to \gamma}{Pgm} \to \frac{\alpha}{Data} \to \frac{\beta \to \gamma}{Pgm}$$

Type of a self-interpreter's meaning:

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

and thus

$$\forall \alpha, \beta \,.\, \texttt{sint} : \frac{\frac{\alpha \to \beta}{Pgm} \to \frac{\alpha}{Univ} \to \frac{\beta}{Univ}}{Pgm}$$

Here $Univ$ **is a universal data type.**

OPTIMALITY AND THE UNIVERSAL DATA TYPE

The optimality criterion: $\mathtt{p}' = [\![\mathtt{spec}]\!]\ \mathtt{sint}\ \mathtt{p}$ **should be as good as** $\mathtt{p}$**.**

Alas this is impossible since:

$$[\![\mathtt{p}]\!] : \alpha \rightarrow \beta$$

but

$$[\![\mathtt{p}']\!] = [\![[\![\mathtt{spec}]\!]\ \mathtt{sint}\ \mathtt{p}]\!] : \frac{\alpha}{Univ} \rightarrow \frac{\beta}{Univ}$$

OPTIMALITY REFORMULATED

A way out: use a self-interpreter with type

$$[\![\mathtt{sint}_{\alpha\to\beta}]\!] : \frac{\alpha\to\beta}{Pgm} \to \alpha \to \beta$$

This can be mechanically obtained from

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

by

$$[\![\mathtt{sint}_{\alpha\to\beta}]\!]\,\mathtt{p}\,\mathtt{a} = decode_{\beta}([\![\mathtt{sint}]\!]\ \mathtt{p}\ encode_{\alpha}(\mathtt{a}))$$

Optimality reformulated: for any $[\![\mathtt{p}]\!] : \alpha\to\beta$ the program

$$\mathtt{p}' = [\![\mathtt{spec}]\!]\ \mathtt{sint}_{\alpha\to\beta}\ \mathtt{p}$$

is at least as fast as $\mathtt{p}$.

OPTIMALITY ACHIEVED

1. L = a first-order call-by-value language with
2. types `unit`, `integer` and sum and product types.
3. The self-interpreter uses a universal type $Univ$.
4. The self-interpreter has been proven correct (Morten Welinder's phd thesis).

OPTIMALITY ACHIEVED

- **Phase 1: specialise using unsophisticated techniques. The output program uses a universal type** $Univ$**.**
- **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.,** η**-reductions for product and sum types.**

Punch line: **It works, and even achieves:**

$$[\![\texttt{spec}]\!]\ \texttt{sint}\ \texttt{sint} =_\alpha \texttt{sint}$$

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?