Types, Partial Evaluation and Optimality

Neil D. Jones
DIKU (Computer Science Department)
University of Copenhagen
DENMARK

Thanks and acknowledgements to

- the TOPPS group at DIKU
- colleagues in many countries

I: Quick review of 1. order partial evaluation

Programs are data objects in a first order data domain D such as:

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

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

$$[\![\_]\!]_{\mathbf{L}} : \mathbf{L}\text{-}programs \to D \rightharpoonup D$$

Program meanings are partial functions:

$$[\![\texttt{p}]\!]_{\mathbf{L}} : D \rightharpoonup D$$

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

$$\begin{array}{lcl} [\![\texttt{(quote ALPHA)}]\!]_{\mathbf{L}} & = & \texttt{ALPHA} \\ [\![\texttt{(lambda (x) (+ x x))}]\!]_{\mathbf{L}}\, 3 & = & \texttt{6} \end{array}$$

Interpreter, compiler, partial evaluator

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

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

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

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

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

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

Techniques for Partial Evaluation

- Applying base functions to known data
- unfolding function calls
- creating one or more *specialized 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))
```

Specialized 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

Suppose $\mathbf{L} = \mathbf{T}$.

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

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

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

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

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

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

Proof. Simple equational reasoning to verify:

$$\begin{array}{lccl} 1. & [\![\texttt{target}]\!](\texttt{d}) & \doteq & [\![\texttt{source}]\!]_{\mathtt{S}}(\texttt{d}) \\ 2. & \texttt{target} & \doteq & [\![\texttt{comp}]\!](\texttt{source}) \\ 3. & \texttt{comp} & \doteq & [\![\texttt{cogen}]\!](\texttt{int}) \end{array}$$

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

Practice: tricky (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.

• **Symbolic composition** of programs `p`, `q`.

Output = program `r`.

Meaning of `r` = (mathematical) composition of the meanings of `p` and `q`.

• **Symbolic specialization** of a function to a known first argument value.

Remainder of this talk

- A notation for the types of symbolic operations. Distinguishes
  - *types of values* from
  - *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.

Types for Symbolic Computation

The *Abstract syntax* of a type t:

$$t\!: \ type \ ::= \ firstorder \mid \underline{type}\,\mathbf{X}$$

$$\mid \quad type \times type \mid type \rightarrow type$$

Type *firstorder* describes values in D.

For each language $\mathbf{X}$ and type t we have a type constructor

$$\underline{t}\,\mathbf{X}$$

Meaning: the set of all $\mathbf{X}$-programs that denote values of type t.

Examples

- Atom `ALPHA` has type *firstorder*
- Lisp program `(quote ALPHA)` has type

$$\underline{firstorder}\,\mathbf{Lisp}$$

Meanings of Type Expressions

The meaning of type expression t is $[\![t]\!]$:

$$
\begin{aligned}
[\![\mathit{firstorder}]\!] &= D \\
[\![t_1 \rightarrow t_2]\!] &= [[\![t_1]\!] \rightarrow [\![t_2]\!]] \\
[\![t_1 \times t_2]\!] &= \{(t_1, t_2) \mid t_1 \in [\![t_1]\!], t_2 \in [\![t_2]\!]\} \\
[\![\, \underline{t}\, \mathbf{X} \,]\!] &= \{\ \mathtt{p} \in D \quad \mid \quad [\![\mathtt{p}]\!]\mathbf{X} \in [\![t]\!]\ \}
\end{aligned}
$$

Some type inference rules:

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

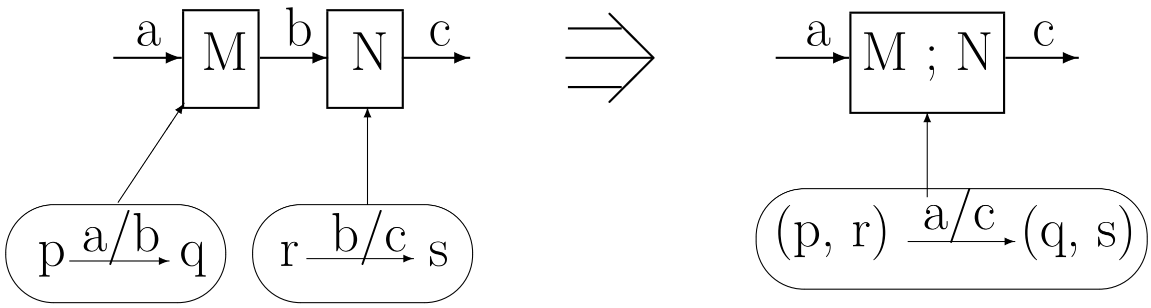
$$
\frac{}{\mathit{firstordervalue} : \mathit{firstorder}}
\qquad
\frac{\mathit{exp} : \underline{t}\, \mathbf{X}}{\mathit{exp} : \mathit{firstorder}}
$$

Symbolic Composition

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

$\underline{(\alpha \to \beta)}$ = 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 $\boxed{oneto\ ;\ squares\ ;\ sum}$

where

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

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 [] 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

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

A Better Definition of Partial Evaluation

Type in the diagram:

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

First Curry:

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

Then generalize:

$$\mathbf{mix} : \forall\alpha\,.\,\forall\tau\,.\,\underline{\alpha \rightarrow \tau} \rightarrow \alpha \rightarrow \underline{\tau}$$

Usually α must be first order.

Definition. Program `mix` $\in D$ is a *partial evaluator* if for all `p, s` $\in D$,

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

Interpreters, compilers, etc. revisited

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

$$[\![\texttt{source}]\!]_{\mathbf{S}} \doteq [\![\texttt{int}]\!]\texttt{source}$$

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

$$[\![\texttt{source}]\!]_{\mathbf{S}} \doteq [\![[\![\texttt{comp}]\!]\texttt{source}]\!]_{\mathbf{T}}$$

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

$$[\![\texttt{p}]\!]\,\texttt{s} \doteq [\![[\![\texttt{mix}]\!]\,\texttt{p}\,\texttt{s}]\!]$$

Type Inference for Self-Application

The Futamura projections:

$$\llbracket \texttt{mix} \rrbracket\, \texttt{int source} \stackrel{def}{=} \texttt{target}$$
$$\llbracket \texttt{mix} \rrbracket\, \texttt{mix int} \stackrel{def}{=} \texttt{compiler}$$
$$\llbracket \texttt{mix} \rrbracket\, \texttt{mix mix} \stackrel{def}{=} \texttt{cogen}$$

Do these type-check?

Recall our type inference rules:

$$\frac{exp_1 : t_2 \rightarrow t_1, \quad exp_2 : t_2}{exp_1\, exp_2 : t_1} \qquad \frac{exp : \underline{t}\, \mathbf{X}}{\llbracket exp \rrbracket_{\mathbf{X}} : t}$$

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

Types of interpreters, etc.

1. Type of `source` : $\underline{\tau}\,\mathbf{S}$

2. Type of $[\![$`int`$]\!]$: $\forall \tau \,.\, \underline{\tau}\,\mathbf{S} \to \tau$

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

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

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

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

and similarly for `source`, `int`, `mix`.

Types during Compilation

We wish to find the type of

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

Assume program `source` has type $\underline{\tau}\,\mathbf{S}$. A deduction:

$$\dfrac{\dfrac{\dfrac{}{[\![\texttt{mix}]\!] : \underline{\rho \rightarrow \sigma} \rightarrow \rho \rightarrow \underline{\sigma}}}{[\![\texttt{mix}]\!] : \underline{\underline{\tau}\,\mathbf{S} \rightarrow \tau} \rightarrow \underline{\tau}\,\mathbf{S} \rightarrow \underline{\tau}} \quad \underline{\texttt{int} : \underline{\tau}\,\mathbf{S} \rightarrow \tau}}{[\![\texttt{mix}]\!]\,\texttt{int} : \underline{\tau}\,\mathbf{S} \rightarrow \underline{\tau}} \quad \texttt{source} : \underline{\tau}\,\mathbf{S}}{[\![\texttt{mix}]\!]\,\texttt{int}\,\texttt{source} : \underline{\tau}}$$

Thus `target` has type $\underline{\tau} = \underline{\tau}\,\mathbf{L}$ (as expected).

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

Types during Compiler Generation: 1

Recall that:

$$\mathtt{compiler} \stackrel{def}{=} \llbracket \mathtt{mix} \rrbracket\, \mathtt{mix}\, \mathtt{int}$$

where interpreter $\mathtt{int}$ has type $\forall \tau \,.\, \underline{\underline{\tau}\,\mathbf{S} \to \tau}$.

We show: If $\mathtt{p}$ has type $\underline{\alpha \to \beta}$

then $\llbracket \mathtt{mix} \rrbracket\, \mathtt{mix}\, \mathtt{p}$ has type $\underline{\alpha \to \underline{\beta}}$

Deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\llbracket \mathtt{mix} \rrbracket : \underline{\rho \to \sigma} \to \rho \to \underline{\sigma}}}{\llbracket \mathtt{mix} \rrbracket : \underline{\underline{\alpha \to \beta} \to \alpha \to \underline{\beta}} \to \underline{\alpha \to \beta} \to \underline{\alpha \to \underline{\beta}}} \quad \mathtt{mix} : \underline{\underline{\alpha \to \beta} \to \alpha \to \underline{\beta}}}{\llbracket \mathtt{mix} \rrbracket\, \mathtt{mix} : \underline{\alpha \to \beta} \to \underline{\alpha \to \underline{\beta}}} \quad \mathtt{p} : \underline{\alpha \to \beta}}{\llbracket \mathtt{mix} \rrbracket\, \mathtt{mix}\, \mathtt{p} : \underline{\alpha \to \underline{\beta}}}$$

Types during Compiler Generation: 2

Recall that:

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

where interpreter $\texttt{int}$ has type $\forall\tau\ .\ \underline{\underline{\tau}_{\mathbf{S}} \rightarrow \tau}$.

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

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

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

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

and so (as desired)

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

Furthermore τ was arbitrary, so

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

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 `sint` is a self-interpreter and `p`, `p′` are programs such that

$$\texttt{p}' = [\![\texttt{mix}]\!]\ \texttt{sint}\ \texttt{p}$$

Correctness of `mix` implies

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

but `p`, `p′` need not be the same programs.

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

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

Intuitively: `mix` has

removed an entire layer of interpretation.

Techniques for Partial Evaluation

- Applying base functions to known data
- unfolding function calls
- creating one or more *specialized functions*

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))
```

Specialized 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 specialized versions of function `a`.

This 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

Trick: split environment into two parallel lists:

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

Part of 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)
...
```

Binding times: `exp,ns,pgm` are *static*,
while `vs` is *dynamic*.

Consequence: all functions in $\texttt{p}' = [\![\texttt{mix}]\!]\ \texttt{sint p}$
have form:

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

Only *one* argument in each $\texttt{p}'$ function ?
This *cannot* be optimal, i.e., as fast as `p` !

Inherited limits during specialisation

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

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

has $k' \leq k$ arguments, if `sint` function `f` has k arguments.

Thus no function in $\texttt{p}'$ has more than k arguments(!)

For interpreter function `eval`, this problem can be solved by *variable splitting*.

Observation: for a fixed `p`, the interpreter's variable `vs` always has *a constant length k*.

Technical solution:

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

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

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

For the "optimal" `mix`, $\texttt{p}' = [\![\texttt{mix}]\!]\ \texttt{sint}\ \texttt{p}$ 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

$$[\![\mathtt{p}]\!] : \mathcal{N} \to \mathcal{N}$$

Then specialized program has a different type:

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

and is significantly less efficient.

The problem is even worse with higher-order types.

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 specializer:

$$[\![\texttt{mix}]\!] : \mathit{Pgm} \rightarrow \mathit{Data} \rightarrow \mathit{Pgm}$$

Correct, but "doesn't tell the whole story"

To clarify the problem, extend $\underline{\alpha \rightarrow \beta}_{\mathbf{L}}$ to

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

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

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

encodings of all values of type α.

Optimality revisited

Type of a specializer's meaning, redone:

$$[\![\texttt{mix}]\!] : \frac{\alpha\rightarrow\beta\rightarrow\gamma}{Pgm} \rightarrow \frac{\alpha}{Data} \rightarrow \frac{\beta\rightarrow\gamma}{Pgm}$$

Type of a self-interpreter's meaning:

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

and thus

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

Here *Univ* is a universal data-type.

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

Alas this is impossible since:

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

but

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

Optimality reformulated

Way out: use a self-interpreter with type

$$[\![\texttt{sint}_{\alpha\longrightarrow\beta}]\!] : \frac{\alpha\longrightarrow\beta}{Pgm} \rightarrow \alpha \rightarrow \beta$$

This can be obtained from

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

mechanically:

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

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

$$\texttt{p}' = [\![\texttt{mix}]\!]\ \texttt{sint}_{\alpha\longrightarrow\beta}\ \texttt{p}$$

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

Optimality achieved

1. **L** = a first-order call-by-value language with
2. types `unit`, `integer` and sum and product types.
3. Self-interpreter uses a universal type *Univ*.
4. Self-interpreter proven correct (Morten Welinder's phd thesis).
5. Phase 1: specialise using unsophisticated techniques.

   The output program uses universal type *Univ*.
6. Phase 2: *Retype* output program, using
   - Type *erasure analysis* that uses
   - non-standard type inference for
   - types that are infinite *regular trees*.
7. Phase 3: an *Identity elimination* phase, e.g., η-reductions for product and sum types.

Punch line: It works, and even achieves:

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

Conclusions

Contributions:

- A notation for the types of symbolic operations. Distinguishes *types of values* from *types of program texts*.
- Natural definitions of type correctness of an interpreter or compiler.
- Makholm: type notation (after some refinement) contributed to solving a long-standing open problem.

More to do:

- Better mathematical understanding of the underbar types.
- *How to prove* that an interpreter or compiler has the desired type?