

---

# Partial Evaluation and Normalisation by Traversals

---

Work in progress by:

▶ **Daniil Berezun**

Saint Petersburg State University

▶ **Neil D. Jones**

DIKU, University of Copenhagen (prof. emeritus)

# INTRODUCTION

---

The much-studied game semantics for PCF can be thought of as a PCF interpreter.

Ong [?] shows that

a  $\lambda$ -expression  $M$  can be evaluated (i.e. normalised) by the algorithm that constructs a traversal of  $M$ .

A *traversal* is a sequence of

- ▶ **subexpressions** of  $M$ . This is a finite set, whose elements we will call **tokens** (think:  $M$  = program, tokens = program points)
- ▶ Any token may have a **back pointers**.

With this approach to normalisation: there is *no need for  $\beta$ -reduction, environments, “thunks” or “closures”* to do the evaluation

# START POINT

---

- ▶ A view of the Oxford normalisation procedure (ONP for short): It is  
an interpreter for  $\lambda$ -expressions
- ▶ ONP builds a set of traversal  $\mathcal{T}\mathbf{rav}(M)$ 
  - Let  $tr = t_0 \bullet t_1 \bullet \dots \bullet t_n \in \mathcal{T}\mathbf{rav}(M)$  where  $t_i$  is a token
  - Syntax-directed inference rules:  
based on syntax of the end-token  $t_n$
  - Action: add 0, 1 or more extensions of  $tr$  to  $\mathcal{T}\mathbf{rav}(M)$ . For each,
    - \* Add a new token  $t'$ , yielding  $tr \bullet t'$
    - \* Add a back pointer from  $t'$

## Data types:

- $tr \in \mathbf{Tr} = \mathbf{Item}^*$
- $\mathbf{Item} = \mathit{subexpression}(M) \times \mathbf{Tr}$

# SOME ONP CHARACTERISTICS

---

## Oxford normalization procedure

- ▶ Applies to **simply-typed**  $\lambda$ -expressions
- ▶ Begins by translating  $M$  into  $\eta$ -long form
- ▶ Realises the complete head linear reduction of  $M$
- ▶ **Correctness:** by game semantics and categories, using  $M$ 's types
- ▶ ONP uses no  $\beta$ -reduction: all is based on subexpressions of  $M$
- ▶ While running, ONP does not use the types of  $M$  at all

## Goals of this research:

- ▶ Extend **ONP** to **UNP**, for the *untyped* lambda calculus
- ▶ **Partially evaluate** a normaliser with respect to "static" input  $M$ .  
Use this to **compile**  $\lambda$ -calculus into a *low-level language*.

# PARTIAL EVALUATION, BRIEFLY

---

A partial evaluator is a **program specialiser**. Defining property of *spec*:

$$\forall p \in \text{Programs} . \forall s, d \in \text{Data} . \llbracket \llbracket \text{spec} \rrbracket (p, s) \rrbracket (d) = \llbracket p \rrbracket (s, d)$$

► Given program  $p$  and **static** data  $s$ , *spec* builds a **residual program**

$$p_s \stackrel{\text{def}}{=} \llbracket \text{spec} \rrbracket p s$$

► **Staging transformation:**

- $\llbracket p \rrbracket (s, d)$  is a **1 stage** computation
  - $\llbracket \llbracket \text{spec} \rrbracket (p, s) \rrbracket (d)$  is a **2 stage** computation
- Program speedup by **precomputation**. Applications: **compiling**, and **compiler generation** (from an interpreter, and by self-applying *spec*).

# WHY PARTIALLY EVALUATE NP

---

1. The *spec* equation for a normaliser program  $\boxed{\text{NP} : \Lambda \rightarrow Traversals}$

$$\boxed{\forall M \in \Lambda . \llbracket \llbracket spec \rrbracket (\text{NP}, M) \rrbracket () = \llbracket \text{NP} \rrbracket (M)}$$

2.  $\lambda$ -calculus tradition:  $M$  is self-contained.

So **why break normalisation into 2 stages?**

(a) The specialised output  $\text{NP}_M = \llbracket spec \rrbracket (\text{NP}, M)$  can be in a **much simpler language** than  $\lambda$ -calculus.

Our candidate is some **low-level language, LLL**.

(b) 2 stages will be natural for **semantics-directed compiler generation**.

LLL can be an intermediate language to express semantics:

►  $\text{NP}_1 = \llbracket spec \rrbracket \text{NP } M : \Lambda \rightarrow LLL$

►  $\text{NP}_2 = \llbracket - \rrbracket : LLL \rightarrow Traversals$  **a semantic function**

# HOW TO PARTIALLY EVALUATE ONP

---

1. **Annotate** parts of normalization procedure as either **static** or **dynamic**.  
Variables ranging over
  - (a) **tokens** are **static**, (subexpressions of  $M$ ; **finitely many**);
  - (b) **back pointers** are **dynamic**; (**unboundedly many**)
  - (c) so the **traversal** is **dynamic** too.
2. Computations in normalization procedure NP are either **unfolded** or **residualised** (runtime code is generated to **do them later**)
  - ▶ Perform **fully static** computations **at partial evaluation time**.
  - ▶ Operations to build or test a traversal: generate **residual code**.

# THE RESIDUAL PROGRAM $\text{ONP}_M = \llbracket \text{spec} \rrbracket \text{ONP } M$

---

ONP is not quite structurally inductive but it is **semi-compositional**:

**Any recursive ONP call has a substructure of  $M$  as argument.**

**Consequences:**

- ▶ The partial evaluator can do, at specialisation time,  
all of the ONP operations that depend only on  $M$
- ▶  $\text{ONP}_M$  performs no operations at all on lambda expressions
- ▶ A specialised program  $\text{ONP}_M$  contains “residual code”:
  - operations to extend the traversal
  - operations to follow back pointers
- ▶ Subexpressions of  $M$  will appear, but are only used as **tokens**:  
Tokens are **indivisible**: used as labels and for equality comparisons



# STATUS: OUR WORK ON SIMPLY-TYPED $\lambda$ -calculus

---

1. We have one version of ONP in `HASKELL` and another in `SCHEME`
2. `HASKELL` version includes: **typing; conversion to eta-long form; the traversal algorithm itself; and construction of the normalised term.**
3. `SCHEME` version: nearly ready to apply automatic partial evaluation.  
Plan: use the `UNMIX` partial evaluator (Sergei Romanenko).
4. The LLL output size is only **linearly larger** than  $M$ ,  $|p_M| = O(|M|)$
5. We have also have a handwritten a ***generating extension*** of ONP.

If  $p_M = \llbracket \text{ONP-gen} \rrbracket^{\text{scheme}}(M)$  then  $\forall M . \llbracket M \rrbracket^\Lambda = \llbracket p_M \rrbracket^{\text{LLL}}$

**Now: LLL = is a tiny subset of `scheme`, so the output  $p_M$  is a `SCHEME` program.**

# MORE WORK FOR SIMPLY-TYPED $\lambda$ -calculus

---

Next steps:

- ▶ Produce a generating extension, **automatically**, by **specialising the specialiser to a  $\Lambda$ -traverser**, using UNMIX
- ▶ Redefine LLL formally as a clean stand-alone subset of HASKELL
- ▶ Use haskell supersompiler
- ▶ Extend existing approach to **programs with input dynamic data**

## STATUS: OUR WORK ON THE UNTYPED $\lambda$ -calculus

---

1. *UNP* is a normaliser for  $\Lambda^{untyped}$ .
2. *UNP* has been done in `HASKELL` and works on a variety of examples.
3. Some of traversal items may have **two back pointers**, in comparison: *ONP* uses only one.
4. As *ONP*, *UNP* is also defined **semi-compositionally** by recursion on syntax of  $\lambda$ -expression *M*.
5. By specialising *UNP*, an **arbitrary untyped  $\lambda$ -expression** can be translated to *LLL*.
6. Correctness proof: pending.

# TOWARDS SEPARATING PROGRAMS FROM DATA IN $\Lambda$

---

Also we have a one more direction for research:

1. An idea is to regard a **computation of  $\lambda$ -expression  $M$  on input  $d$**  as a **two-player game between the LLL-codes for  $M$  and  $d$** .
2. An interesting examples in this case is a usual  $\lambda$ -calculus definition of function *mult* on Church numerals.
3. Amazing fact is that Loops from out of nowhere:
  - ▶ **Neither `mul` nor the data contain loops**;
  - ▶ But `mul` function is compiled into **an LLL-program with two nested loops**;
  - ▶ We also expect that we can do the computation **entirely without back pointers**.
4. Right now we are trying to express such program-data games in a **communicating** version of *LLL*.

# REFENERCES

---