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 λ-expression M* can be evaluated by the algorithm that constructs a traversals of *M

A *traversal* is a sequence of

- **tokens, subexpressions of *M***

  (think: M = program, tokens = program points)

- **Any token may have a back pointer**

- **Oxford normalization procedure (*ONP*)**

  - **Constructs $\mathfrak{Trav}(M)$**

START POINT

- **An operational view of the Oxford normalisation procedure:**

  interpreter for λ-expressions

  - **Let $tr = t_0 \bullet t_1 \bullet \cdots \bullet t_n \in \mathfrak{Trav}(M)$ where t_i is a token**
  - **Syntax-directed inference rules:**

    based on syntax of the end-tokens t_n

  - **Action: add 0, 1 or more extensions of tr to $\mathfrak{Trav}(M)$. For each,**
    - **Add a new token t', yielding $tr \bullet t'$**
    - **Add a back pointer from t'**

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

SOME *ONP* CHARACTERISTICS

Oxford normalization procedure

- **Applies to simply-typed λ-expressions**
- **Begins by translating M into η-long form**
- **Realises the compete head linear reduction of M**
- **Correctness: by game semantics and categories, using M's types**
- **ONP uses no β-reduction, environments, . . .**
- **While running, ONP does not use the types of M at all**

Goals of this research:

- **Extend *ONP* to *UNP*, for the *untyped* λ-calculus**
- **Partially evaluate a normaliser**

PARTIAL EVALUATION, BRIEFLY

Partial Evaluation = program specialization. Defining property of $spec$:

$$\forall p \in Programs\ .\ \forall s, d \in Data\ .\ [\![[\![spec]\!](p, s)]\!](d) = [\![p]\!](s, d)$$

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

$$p_s \stackrel{def}{=} [\![spec]\!]\ p\ s$$

- **Program speedup by precomputation**
- **Staging transformation:**
  - $[\![p]\!](s, d)$ **is a 1 stage computation**
  - $[\![[\![spec]\!](p, s)]\!](d)$ **is a 2 stage computation**
- **Applications: compiling, and compiler generation**
- **An old idea: Semantics directed compiler generation**

WHY PARTIALLY EVALUATE NP

1. The $spec$ equation for a normaliser program $\boxed{\mathbf{NP} : \Lambda \to Traversals}$

$$\boxed{\forall M \in \Lambda \,.\, [\![\, [\![spec]\!](\mathbf{NP}, M)]\!]() = [\![\mathbf{NP}]\!](M)}$$

2. λ-calculus tradition: M is self-contained

   Why break normalisation into 2 stages?

   (a) The specialized output $\mathbf{NP}_M = [\![spec]\!](\mathbf{NP}, M)$ can be in a much simplier language than λ-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:

   - $\mathbf{NP}_1 = [\![spec]\!]\ \mathbf{NP}\ \boldsymbol{M} : \Lambda \to LLL$
   - $\mathbf{NP}_2 = [\![\_]\!] : LLL \to Traversals$ a semantic function

HOW TO PARTIALLY EVALUATE *ONP*

1. **Annotate** parts of normalization procedure as either **static** or **dynamic**
   (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 *ONP* are either **unfolded** or **residualised**
   - Perform **fully static** computations **at partial evaluation time**
   - Operations to build or test a traversal: generate **residual code**

THE RESIDUAL PROGRAM $ONP_M = [\![spec]\!]\ ONP\ M$

ONP is **semi-compositional**:

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

Consequences:

- The partial evaluator can do (at specialization time)

  all of the *ONP* operations that depend only on M

- ONP_M performs no operations at all on λ-expressions

- A specialized program 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 λ-calculus

1. **We have one version of ONP in HASKELL and another in SCHEME**
2. **SCHEME version: nearly ready to apply automatic partial evaluation**
   Plan: use the UNMIX partial evaluator (Sergei Romanenko)
3. **The *LLL* program are only linearly larger than M, $|p_M| = O(|M|)$**
4. **Handwritten a *generating extension* of *ONP***

   If $p_M = [\![\text{ONP-gen}]\!]^{Scheme}(M)$ then $\forall M \,.\, [\![M]\!]^{\Lambda} = [\![p_M]\!]^{LLL}$

5. **Next steps:**

   - **Produce a generating extension, automatically, using UNMIX**
   - **Redefine *LLL* formally as a *clean stand-alone* subset of HASKELL**
   - **Use HASKELL supercompiler**
   - **Extend existing approach to programs with dynamic input**

STATUS: OUR WORK ON THE UNTYPED λ-calculus

1. ***UNP*** **is a normaliser for** Λ

2. ***UNP*** **has been done in** HASKELL **and works on a variety of examples**

3. **Some 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 λ**-expression** M

5. **By specializing** ***UNP***, **an arbitrary untyped** λ**-expression can be translated to** ***LLL***

6. **Correctness proof: pending**

TOWARDS SEPARATING PROGRAMS FROM DATA IN Λ

One more research direction:

- **An idea: consider a computation of λ-expression M on input d as a two-player game between the *LLL* -codes for M and d**
- **An interesting example in this case a is usual λ-calculus definition of function $mult$ on Church numerals**
  - **Amaizing fact: loops come from out of nowhere**
  - **We also expect that we can do the computation (in this case) entirely without back pointers**
- ***Communicating* version of *LLL*.**

REFERENCES