Partial Evaluation and Normalisation by Traversals

Work in progress by:

- **Daniil Berezun**
  State University of St. Petersburg
- **Neil D. Jones**
  DIKU, University of Copenhagen (prof. emeritus)

A BELATED OBSERVATION (LAST YEAR)

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

Ong [1] shows that

A λ-expression M can be evaluated (normalised) by an 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
- any token in a traversal may have a back pointer (aka. justifier).

Consequence: there is *no need for β-reduction, environments, "thunks" or "closures"* to do the evaluation(!)

Root: research on full abstraction for PCF.

START OF THIS WORK

Ong's normalisation procedure (ONP for short) can be seen as

an interpreter for λ-expressions

- ONP systematically builds a set of traversals $\mathfrak{Trav}(M)$. How?
- Traversal : $tr = t_0 \ldots \cdot t$ where t is a token (subexpression of M)
- Syntax-directed inference rules: based on syntax of the end-token t
- Action: add 0, 1 or more extensions of tr to $\mathfrak{Trav}(M)$. For each,
  - Add a new token t', yielding $tr \cdot t'$
  - Add a back pointer from t' (or no back pointer, it depends on t)

SOME CHARACTERISTICS

Ong's normalisation procedure

- Applies to simply-typed λ-expressions
- Begins by translating M into η-long form
- Realises the head linear reduction of M, one step at a time
- Correctness proof: by game semantics and categorical reasoning, strongly based on the type structure of M

Operational observations:

No β-reduction: all is based on subexpressions of M!

While running, the ONP algorithm does not use the types of M at all

Narural idea: partially evaluate normaliser ONP with respect to M

PARTIAL EVALUATION, BRIEFLY

A partial evaluator is a program specialiser. 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* $p_s \stackrel{def}{=} [\![spec]\!] p\ s$.**
- **When run on any remaining "dynamic" data d, residual program p_s computes what p would have computed on both data inputs s and d.**
- **The net effect: a *staging transformation*: $[\![p]\!] s\ d$ describes a 1 stage computation, while $[\![[\![spec]\!] p\ s]\!]\ d$ describes computation in 2 stages.**

  It makes sense even if s or d are empty.
- **Well-known in recursive function theory, as the S-1-1 theorem.**
- **Partial evaluation = engineering contruction for the S-1-1 theorem.**
- **Program speedup by precomputation. Applications: compiling, and compiler generation (from an interpreter, and by self-applying $spec$).**

COULD NORMALISATION BE STAGED?

1. The S-1-0 equation for the ONP program:

$$\boxed{\forall M \in \Lambda \,.\, [\![\, [\![spec]\!]\ \text{ONP}\ M]\!] \;=\; [\![\text{ONP}]\!]\ M}$$

2. There is no external dynamic data, as M is self-contained.

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

   (a) Specialiser output $\text{ONP}_M = [\![spec]\!]\ \text{ONP}\ M$ is naturally in a **much simpler language** than the λ-calculus.
   Our candidate: LLL, a "low-level language".

   (b) Planned extension: Think about S-1-1: **computational complexity** of normalising if M is applied to an **external input** d at run-time.

$$\boxed{\forall M \in \Lambda, d \in D \,.\, [\![\, [\![spec]\!]\ \text{ONP}\ M]\!](d) \;=\; [\![\text{ONP}]\!]\ (M\ d)}$$

   (c) 2 stages will be natural for ***semantics-directed compiler generation***.
   Use LLL as intermediate language to express semantics.

ANOTHER WAY TO SAY IT

Given M, we **factor** its traversal algorithm:

$$\text{ONP} : \Lambda \to \mathit{Traversals}$$

into two stages:

$$\text{ONP}_1 : \Lambda \to \text{LLL-pgms} \quad \text{and} \quad \text{ONP}_2 : \text{LLL-pgms} \to \mathit{Traversals}$$

where

- $\text{ONP}_1 = [\![spec]\!]\ \text{ONP}\ M$

  An LLL program; result of partially evaluating ONP w.r.t. input M

- $\text{ONP}_2 = [\![\ \_\ ]\!]^{LLL}$ the semantic function of LLL-programs

HOW TO PARTIALLY EVALUATE ONP WITH RESPECT TO M?

1. Write ONP as a program.

2. Annotate parts of ONP as either static or dynamic.

   Data:

   (a) tokens, i.e., λ-expressions (each is a subexpression of M);

   (b) back pointers;

   (c) the traversal being built

3. Classify data 2a as static (there are only finitely many)

4. Classify data 2b, 2c as dynamic

5. Recursive calls within ONP:

   - Call to a smaller λ-expression: static
   - Any other call: dynamic

   Unfold at Partial eval. time, else make the call residual

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

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

Any recursive ONP call has $\underline{\underline{\text{a substructure of } M}}$ as argument.

Consequences:

- The partial evaluator can perform, at specialisation time,
  all of the ONP operations that depend only on M
- So ONP$_M$ performs no operations at all on lambda expressions.
- ONP$_M$ contains operations to build the traversal (and to follow back pointers when needed to do this)
- Subexpressions of M will appear, but are only used as tokens: they are indivisible, only used for equality comparisons with other tokens

EXAMPLE: ONP SPECIALISED TO $M = \lambda nsz \,.\, s(nsz)$

```
Eta-long form:  \ n s z . s (\ . n (\ q . s (\ . q)) (\ . z))

Tree for eta-long form:
(define succ
'(A :lambda (n s z)
  (B : s
     ((C :lambda () (D : n ((E :lambda (q) (F : s ((G :lambda () (H : q ()))))))
                            (I :lambda () (J : z ())))))))))

Residual code to add traversal items
====================================

(reverse (A '((A 0))))               ; MAIN function: call A

(define (A t) (B (cons (list 'B (FQ 'A t)) t)))
(define (B t) (CGOTO t 2))                     ; activate s
(define (C t) (D (cons (list 'D (FQ 'A t)) t)))
(define (D t) (CGOTO t 1))                     ; activate n
(define (E t) (F (cons (list 'F (FQ 'A t)) t)))
(define (F t) (CGOTO t 2))                     ; activate s
(define (G t) (H (cons (list 'H (FQ 'E t)) t)))
(define (H t) (CGOTO t 1))                     ; activate q
(define (I t) (J (cons (list 'J (FQ 'A t)) t)))   ;(long form!)
(define (J t) (CGOTO t 3))                     ; activate z

  ;   (CGOTO t i) = computed goto: activate i-th argument of an APPLY
```

THE REST: GOTO AND BACKPOINTER SEARCH FCNS

```
(define (CGOTO t i)     ;  p :=  the relevant "APPLY" from traversal t
      (let ((p (- (cadar t) 1))) (CGOTO_0 (caar (pfx p t)) t p i)))

(define (CGOTO_0 have t p i)                         ; Branch to find the relevant "APPLY"
    (if (equal? have 'B)     ; The relevant "APPLY" is s(n s z)
      (if (equal? i 1) (C (cons (list 'C p) t)) (error 'goto:error))
      (if (equal? have 'D)  ; The relevant "APPLY" is n s z (2 arguments)
        (if (equal? i 2)
          (I (cons (list 'I p) t))
          (if (equal? i 1) (E (cons (list 'E p) t)) (error 'goto:error)))
        (if (equal? have 'F)                         ; The relevant "APPLY" is s(\ . q)
          (if (equal? i 1) (G (cons (list 'G p) t)) (error 'goto:error))
          'ERROR))))

  (define (FQ abs t)  ;  Back-chain to find the λ binder of "abs" in the traversal t
    (letrec ((f
              (lambda (t0)
                (if (equal? abs (caar t0))
                  (length t0)
                  (let ((bp (cadar t0))) (f (pfx (- bp 1) t)))))))
      (f t)))

  (define (pfx n t) . -- length n prefix of t --
```

STATUS: OUR WORK ON SIMPLY-TYPED λ-calculus

1. One ONP version 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 UNMIX system (Sergei Romanenko).

4. We have handwritten an LLL-*generating extension* of ONP.

   Symbolically:

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

   Current implementation: the output p_M is a SCHEME program.

5. The LLL output program size is only linearly larger than M, satisfying

   $$|p_M| = O(|M|)$$

MORE TO DO, FOR THE SIMPLY-TYPED λ-calculus

1. Extend the approach to **programs with input data**.

2. Produce a generating extension automatically by **specialising the specialiser to a Λ-traverser**, using UNMIX.

3. Using UNMIX, the programs produced by the generating extension will be in SCHEME.

   - **Practical advantage**: p_M is directly executable (e.g., by RACKET).

   - **Disadvantage**: p_M in this form could be system-dependent.

4. To do: redefine the LLL language formally, e.g., a tiny

   1. order call-by-value language with HASKELL-like syntax.

5. Then: produce programs in LLL instead of SCHEME.

STATUS: OUR WORK ON THE UNTYPED λ-calculus

1. UNP, a normaliser for $\Lambda^{untyped}$, has been written in HASKELL and works on a variety of examples

2. It uses four back pointers (in comparison: ONP uses 2, one for binders and one for arguments).

3. The UNP algorithm is (again) defined by semi-compositional recursion on λ-expression's syntax.

4. By specialising UNP, an arbitrary untyped λ-expression can be translated to LLL.

5. No SCHEME version or generating extension yet.

TOWARDS SEPARATING PROGRAMS FROM DATA IN Λ

1. An idea: regard a computation of λ-expression M on input d as a two-player game between the LLL-codes for M and d.

2. A promising example: `mul` , the usual λ-calculus definition on Church numerals.

3. Loops from out of nowhere:

   - Neither `mul` nor the data contain loops;
   - but `mul` is compilable into an LLL-program with two nested loops. Applied to two Church numerals, it computes their product.
   - Further: computation can be done entirely without back pointers.

4. Current work: design a *communicating* version of LLL to express such program-data games.

   A lead: apply traditional methods for compiling *remote function calls*.

AN OLD DREAM:
SEMANTICS-DIRECTED COMPILER GENERATION

(Just a wild idea for now, needs much more thought and work.)

Idea: specify the semantics of a subject programming language

(e.g., call-by-value λ-calculus, imperative languages, etc.)

by mapping source programs into LLL.

A “gedankeneksperiment”, to get started:

Express the semantics of Λ by semi-compositional semantic rules sans variable environments, thunks, etc.

$$[\![\ ]\!]^{\Lambda} : \Lambda \to \mathrm{LLL}$$

Expectations/hopes:

- Reasonably many programming languages can be specified this way
- Common framework: compiling, optimisation,... are all reduced to questions and algorithms concerning LLL programs

SOME RELATED WORK

References

[1] C.-H. Luke Ong. Normalisation by traversals. *CoRR*,abs/1511.02629, 2015.

[2] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[3] William Blum and Luke Ong. A concrete presentation of game semantics. In *Galop 2008:Games for Logic and Programming Languages*.

[4] R. P. Neatherway, S. J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In *ICFP*, 2012.

[5] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 2000.

[6] Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*. Springer, 1980.

[7] Neil D. Jones and Steven S. Muchnick. The complexity of finite memory programs with recursion. *J. ACM*, 25(2):312–321, 1978.