Compiling Untyped $\lambda$-calculus to Lower-level Code by Game Semantics and Partial Evaluation

▶ Daniil Berezun  
State University of St. Petersburg

▶ Neil D. Jones  
DIKU, University of Copenhagen (prof. emeritus)
Implicit in PCF research (Ong/Abramsky/. . . 1990s)

Explicit in ong [1]:

1. Convert typed \( \lambda \)-expression \( M \) into long form \( M^{lf} \)

2. Traverse the syntax nodes of \( M^{lf} \):

3. Traversal builds a history \( h \) of the normalisation of \( M \)

4. \( h \in H = (\text{Subexp}(M) \times H)^* \)

Origins: research on full abstraction for PCF.
The game semantics for PCF amounts to an executable implementation of PCF, i.e., a PCF interpreter.

An observation: this implementation uses none of the usual machinery: parameters by closures or thunks; bindings by environments. (Instead, all is done by tokens and back pointers).

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)

- each token in a traversal may have a back pointer (aka. justifier).
ONG’S NORMALISATION PROCEDURE  ONP

▶ applies to simply-typed λ-expressions

▶ begins by translating $M$ into $\eta$-long form

▶ effect: head linear reduction of $M$, one step at a time

▶ Correctness: proven by game semantics and category theory. Strongly based on $M$’s types.

Properties of the normalisation procedure:

- Uses no $\beta$-reduction: just take a walk through subexpressions of $M$.

While running, ONP does not use the types of $M$ at all.
OUR WORK

► Extend Ong [1] to the untyped \( \lambda \)-calculus. We use two kinds of back pointers.

► Call the this algorithm \( UNP \). Concretely, \( UNP \) can be programmed in \textsc{haskell} or \textsc{scheme}.

Partial evaluation: we construct low-level code for \( \lambda \)-expression \( M \) by partial evaluation:

\[ \llbracket spec \rrbracket(UNP, M) = \text{Target code for } M \]

► More: one can generate a compiler from \( UNP \) by partial evaluation:

\[ \llbracket cogen \rrbracket(UNP) \in \text{ULC } \Rightarrow \text{LLL} \]
MULTIPLYING CHURCH NUMERALS: $2 \times 2 = 2(2S)Z$

Church numeral for $n : \lambda s \lambda z . s(\cdots(sz)\cdots)$

$mul = \lambda mn sz . m(ns)z$

Normal form of $2 \times 2$: $S(S(S(SZ)))$

\[
\begin{array}{cl}
m = 2 \\
\downarrow \\
\lambda s1 \\
\downarrow \\
\lambda z1 \\
\downarrow \\
\lambda s1 \\
\downarrow \\
\lambda z1 \\
\downarrow \\
\lambda s1 \\
\downarrow \\
z1 \\
\frac{\text{PROGRAM}}{
\begin{array}{cl}
\Leftarrow n = 2 \\
\\shortdownarrow \\
\text{\rule{5cm}{0.5mm}} \\
\end{array}
}
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{z1} \\
\Downarrow \\
\lambda s2 \\
\Downarrow \\
\lambda z2 \\
\Downarrow \\
\lambda s2 \\
\Downarrow \\
z2 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]

\[
\begin{array}{cl}
\frac{\text{\rule{1cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
\lambda z1 \\
\Downarrow \\
\lambda s1 \\
\Downarrow \\
z1 \\
\frac{\text{\rule{5cm}{0.5mm}}}{\text{\rule{1cm}{0.5mm}}} \\
\text{\rule{1cm}{0.5mm}} \\
\end{array}
\]
GAME: DATA $m = 2, n = 2$ VERSUS PROGRAM: STEPS 1–6

1: $\@_1$

$\downarrow$

3: $\lambda s_1$

4: $\lambda z_1$

5: $\@_3$

6: $s_1$

$\downarrow$

$\downarrow$

$\downarrow$

$\downarrow$

$\downarrow$

$\downarrow$

$\downarrow$

$\downarrow$

2: $\@_2$

$\lambda s_2$

$\lambda z_2$

$\@_5$

$\@_6$

$\@_7$

$\rightarrow$ PROGRAM

$\leftarrow$ PROGRAM

$\leftarrow n = 2$
TRAVERSAL OF $2 \times 2 = 2(2S)Z$: STEPS 7–11

$$m = 2$$

$$\Rightarrow$$

$1$: $@_1$

$2$: $@_2$

$7$: $@_5$

$Z$

$S$

$1$: $@_1$

$2$: $@_2$

$3$: $\lambda s1$

$4$: $\lambda z1$

$5$: $@_3$

$6$: $s1$

$7$: $@_4$

$8$: $\lambda s2$

$9$: $\lambda z2$

$10$: $@_6$

$11$: $s2$

$11$: $s2$

$n = 2$
TRAVERSAL OF $2 \times 2 = 2(2S)Z$: STEPS 12–16

$m = 2$

$\lambda s1$

$\lambda z1$

@$3$

@$4$

$\lambda s2$

$\lambda z2$

@$8$

@$9$

$S : 12, 15$

$n = 2$

$\Rightarrow$ PROGRAM

$z1$

@$1$

$\Rightarrow$ PROGRAM

$z2 : 16$

$11: s2$

$14: s2$

$12, 15$
TRAVERSAL OF $2 \ast 2 = \underline{2(2S)}Z$: STEPS 17–18

$m = 2 \Downarrow$

1: $\@_1$

2: $\@_2$

3: $\lambda s_1$

4: $\lambda z_1$

5: $\@_3$

6: $s_1$

7: $\@_5$

8: $\lambda s_2$

9: $\lambda z_2$

10: $\@_6$

11: $s_2$

12: $s_2$

13: $z_2 : 16$

14: $s_2$

15: $S : 12, 15$

16: $z_2 : 16$

17: $\@_4 : 17$

18: $s_1$

PROGRAM

$\Leftarrow n = 2 \Downarrow$
TRAVERSAL OF $2 \times 2 = 2(2S)Z$: STEPS 19–23

$m = 2$

$\downarrow$

$\lambda s_1$

\[ \lambda z_1 \]

$@_3$

$s_1$

$z_1$

\[ @_4 : 17 \]

$@_5$

$S : 12, 15$

\[ n = 2 \]

$\leftarrow$ PROGRAM
TRAVERSAL OF $2 \times 2 = 2(2S)Z$: STEPS 24–30
How on earth did we select the right node visit sequence? There are many possibilities, mostly wrong!

We develop several semantics.

▶ Semantics 1 is classical $\beta$-reduction (a deterministic version)

▶ Semantics 5 resembles Ong’s, with no environments, thunks, etc. but two kinds of back pointers. Leftmost head linear reduction

▶ All traverse subexpressions of $M$ in the same order

All the semantics achieve the canonical traversal order.

How is it defined? Mark the subexpression occurrences in $M$. Then trace their order during the complete leftmost head $\beta$-reduction.
Semantics 1: A classical $\beta$-reduction semantics.

Semantics 2: An environment semantics as in functional programming.

Semantics 3: Environment-based but tail recursive. Realise nested evaluator calls by data structures.

Semantics 4: First history semantics. Implement the control data by back pointers into the computational history.

Semantics 5: Final history semantics. Implement the environments by back pointers into the computational history.

This history records the normaliser calls done until now (with argument values). Net effect: Semantics 5 is

$$\text{UNP} \in \Lambda_L$$

$\text{UNP}$ is a first-order program.
Classical reduction: needs a flag to avoid reducing $e_0$ twice in an application $(\lambda x.e_0)@e_2$.

Environment semantics: $\rho \in Env = Variable \rightarrow Exp \times Env$. Two excerpts:

$$\llbracket x \rrbracket \rho = \text{let } (e_0, \rho_0) = \rho(x) \text{ in } \llbracket e_0 \rrbracket \rho_0$$

$$\llbracket e_1@e_2 \rrbracket \rho = \text{let } (\lambda x.e_0, \rho_0) = \llbracket e_1 \rrbracket \rho \text{ in } \llbracket e_0 \rrbracket \rho_0[x \mapsto (e_2, \rho)]$$

Environment semantics is not compositional, but it is semi-compositional. This means:

in any call $\llbracket e \rrbracket \rho$ that occurs while evaluating $\lambda$-expression $M$, argument $e$ will be a subexpression of $M$.

(This is good for compilation and partial evaluation.)
CONTINUATIONS AND DEFUNCTIONALISATION

Goal: Semantics 3 = tail-recursive version of Semantics 2. Techniques: well-known, e.g. John Reynolds’ **Definitional interpreters** paper.

- **Continuations**: modify Semantics 2 to have linear control flow.

  **Defunctionalisation**: then replace the continuation functions by data structures.

- Example of net effect: replace

  \[
  \left[ e_1 \circ e_2 \right]^2 \rho = \text{let } (\lambda x. e_0, \rho_0) = \left[ e_1 \right]^2 \rho \text{ in } \left[ e_0 \right]^2 \rho_0[x \mapsto (e_2, \rho)]
  \]

  by:

  \[
  \left[ e_1 \circ e_2 \right]^3 \rho k = \left[ e_1 \right]^3 \rho \langle Kapp e_2 \rho \rangle k
  \]

  plus:

  \[
  \text{applycont } \langle Kapp e_2 \rho \rangle e_0 \rho_0 = \left[ e_0 \right]^3 \rho_0[x \mapsto (e_2, \rho)] k
  \]
Semantics 4:

► Replace the continuation argument \( k \) by a history \( h \).

► \( h \) is a accumulative trace that remembers

\[ h \in H = (\text{Exp} \times \text{Env} \times H)^* \]

► What’s the point? We can replace a continuation data structure such as \( \langle Kapp \ e_2 \ \rho \ k \rangle \) by a pointer to the time at which it was created (call it \( t \)).

If you are given a back pointer as value of \( t \), you can find the parts that \( \langle Kapp \ e_2 \ \rho \ k \rangle \) was built from in the history.

► Effect: save the time and space needed to build the continuation data.

► However this has a cost: keeping the history available for access.
Semantics 5:

► Replace the environment $\rho$ in Semantics 4 by a back pointer into the history $h$.

► Same idea, but a separate pointer is needed.

► A difference from Semantics 2-3-4:

  The value of a variable $x$ is found,

  • not by applying a single function $\rho$, but
  • by following a chain of back pointers, to locate the place where $x$ was last bound.

► Effect: all of the normaliser’s arguments are now first-order.
A partial evaluator is a program specialiser. Defining property of $spec$:

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

- Program speedup by precomputation. Applications: compiling, and compiler generation (from an interpreter, and by self-applying $spec$).

- Given program $p$ and “static” data $s$, $spec$ builds a residual program $p_s \overset{def}{=} \llbracket spec \rrbracket (p, s)$.

- When run on any remaining “dynamic” data $d$, residual program $p_s$ computes what $p$ would have computed on both data inputs $s$ and $d$.

- Net effect: a staging transformation: $\llbracket p \rrbracket (s, d)$ is a 1 stage computation; but $\llbracket \llbracket spec \rrbracket (p, s) \rrbracket (d)$ is a 2 stage computation.

- Well-known in recursive function theory, as the $S$-$1$-$1$ theorem.

- Partial evaluation = engineering the $S$-$1$-$1$ theorem on real programs.
LLL is a tiny **tail recursive first-order functional** language. Essentially a machine language with a heap. Functional version of **WHILE** in book: *Computability and Complexity from a Programming Perspective*

**SYNTAX**

```
program ::= f1 x = e1 ... fn x = en

e ::= x | f e
    | token | case e of token1 -> e1 ... tokenn -> en
    | (e,e) | case e of (x,y) -> e
    | [] | case e of [] -> e x:y -> e

x ::= variable

token ::= an atomic symbol (from a fixed alphabet)
```

Variables have **SIMPLE TYPES** (not depending on \(M\)):

```
tau ::= Token | tau x tau | [ tau ]
```

A token, or a product type, has a **static structure**, fixed for any one **LLL** program. A list type \([\tau]\) (dynamic) has constructors \([]\) and \(::\).
HOW TO PARTIALLY EVALUATE NP (IN PROGRAM FORM) WITH RESPECT TO STATIC $\lambda$-EXPRESSION $M$?

1. **Annotate** parts of NP as either **static** or **dynamic**. Variables ranging over
   - (a) **tokens** are static, i.e., $\lambda$-expressions (subexpressions of $M$);
   - (b) **back pointers** are dynamic;
   - (c) so the **traversal** being built is dynamic too.

2. **Classify** data 1a as static (there are only finitely many)

3. **Classify** data 1b, 1c as dynamic (there are unboundedly many)

4. **Computations in NP** are either **unfolded** (done at PE time) or **residualised** (runtime code is generated to do them at stage 2)
   - Perform **fully static computations** at partial evaluation time.
   - Operations to build or test a traversal: generate **residual code**.
THE RESIDUAL PROGRAM $\mathbf{NP}_M = [\mathit{spec}] \mathbf{NP} \; M$

If NP is semi-compositional:

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

Then:

- The partial evaluator can do, at specialisation time, all of the NP operations that depend only on $M$

- $\mathbf{NP}_M$ contains “residual code”:
  - operations to extend the traversal; and
  - operations to follow back pointers

- $\mathbf{NP}_M$ performs no operations at all on lambda expressions (!)

- Subexpressions of $M$ will appear, but are only used as tokens:
  Tokens are indivisible, only used for equality comparisons with other tokens
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 \( \lambda \)-calculus, imperative languages, etc.)
by mapping source programs into LLL.

A “gedankeneksperiment”, to get started:

Express the semantics of \( \Lambda \) by semi-compositional semantic rules without variable environments, thunks, etc:

\[
\llbracket \cdot \rrbracket^\Lambda : \Lambda \to \text{LLL}
\]

Expectations/hopes:
- Reasonably many programming languages can be specified this way
- A generalising framework: compiling, optimisation, . . . tasks can all be reduced to questions and algorithms concerning LLL programs
1. An idea: formalise a computation of $\lambda$-expression $M$ on input $d$ as a two-player game between the LLL-codes for $M$ and $d$.

2. An example: $mul$, usual $\lambda$-calculus definition on Church numerals.

3. Loops appear from out of nowhere:
   - Neither $mul$ nor the data contain loops;
   - but $mul$ is compiled into an LLL-program with two nested loops.
   - Expect: can do the computation entirely without back pointers.

4. Current work: express such program-data games in a communicating version of LLL. A lead: apply traditional methods for compiling remote function calls.

5. Next step: optimise LLL. Remove all inessential bits of the traversal.

6. Think about complexity and data-flow analysis of such programs.
References


