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 [?] 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

  (think: M = program, tokens = program points)
- any token in a traversal may have back pointers (aka. justifiers).

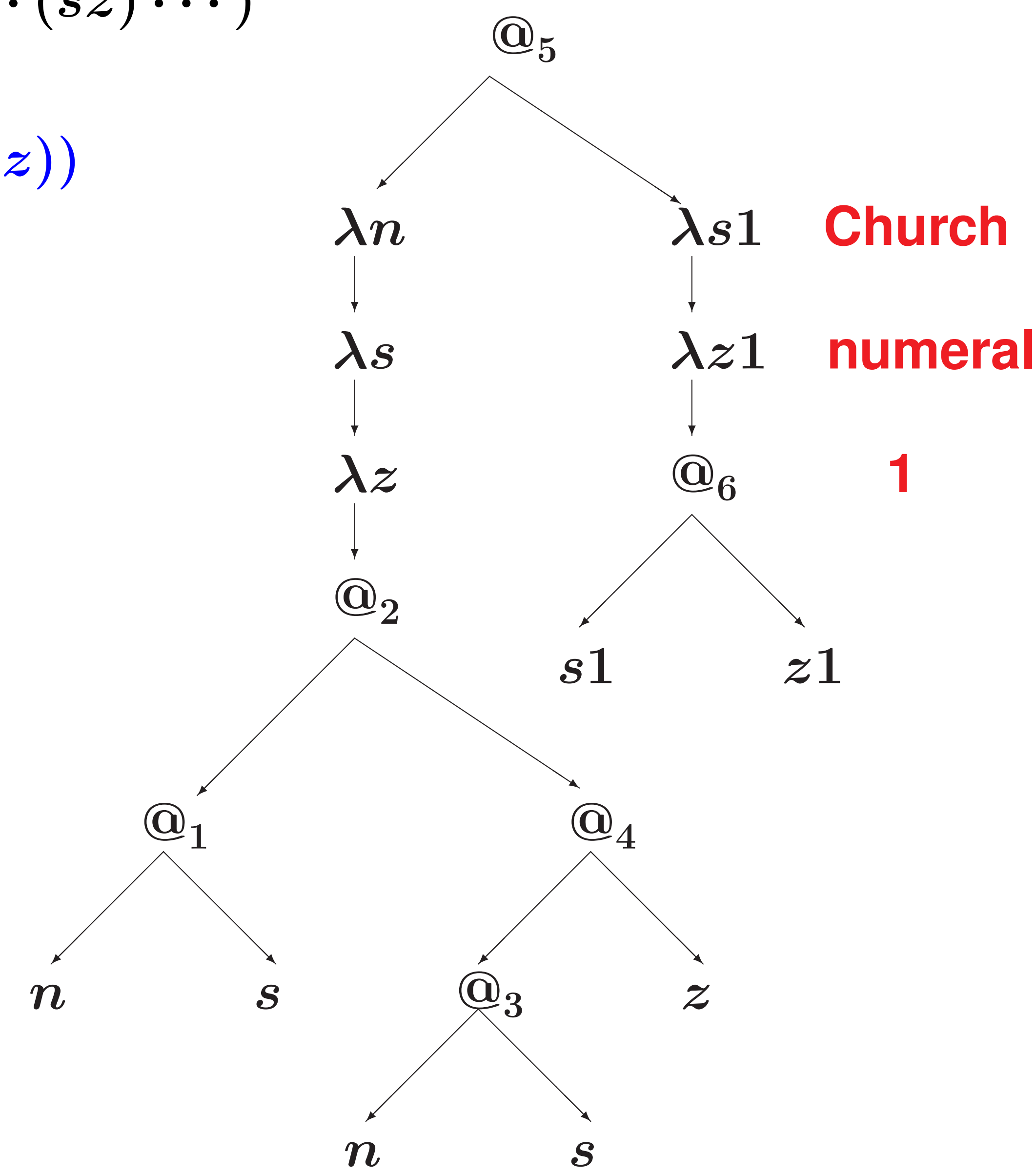
With this approach to normalisation: there is *no need for β-reduction, environments, "thunks" or "closures"* to do the evaluation(!)

Origin: research on full abstraction for PCF.

BACK POINTER MAGIC: DOUBLING A CHURCH NUMERAL

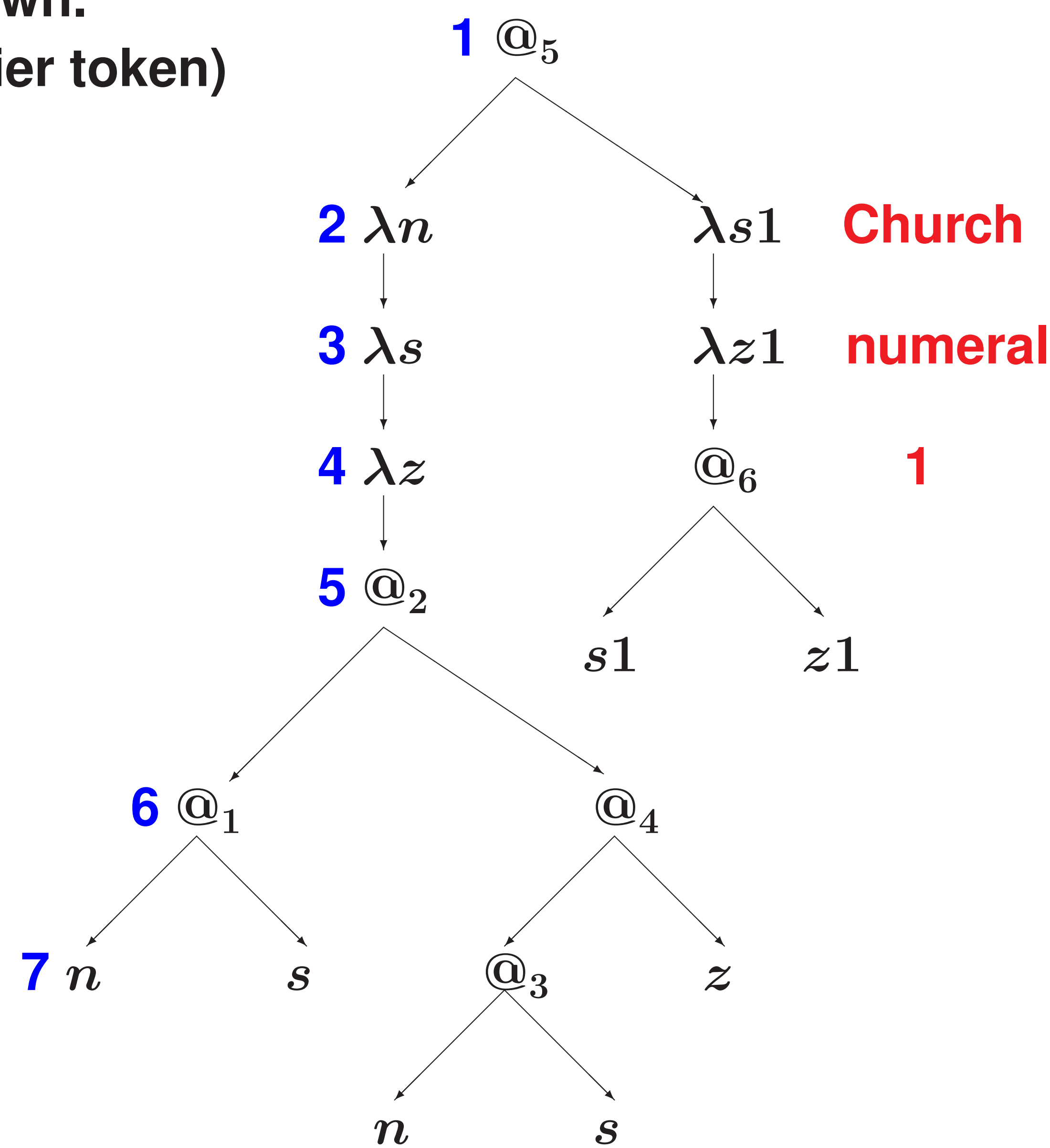
Church numeral for $n : \lambda s \lambda z \,.\, s(\cdots(sz)\cdots)$

$$double = \lambda nsz \,.\, ns(nsz)$$
$$= \lambda n \lambda s \lambda z \,.\, ((n @_1 s) @_2 ((n @_3 s) @_4 z))$$



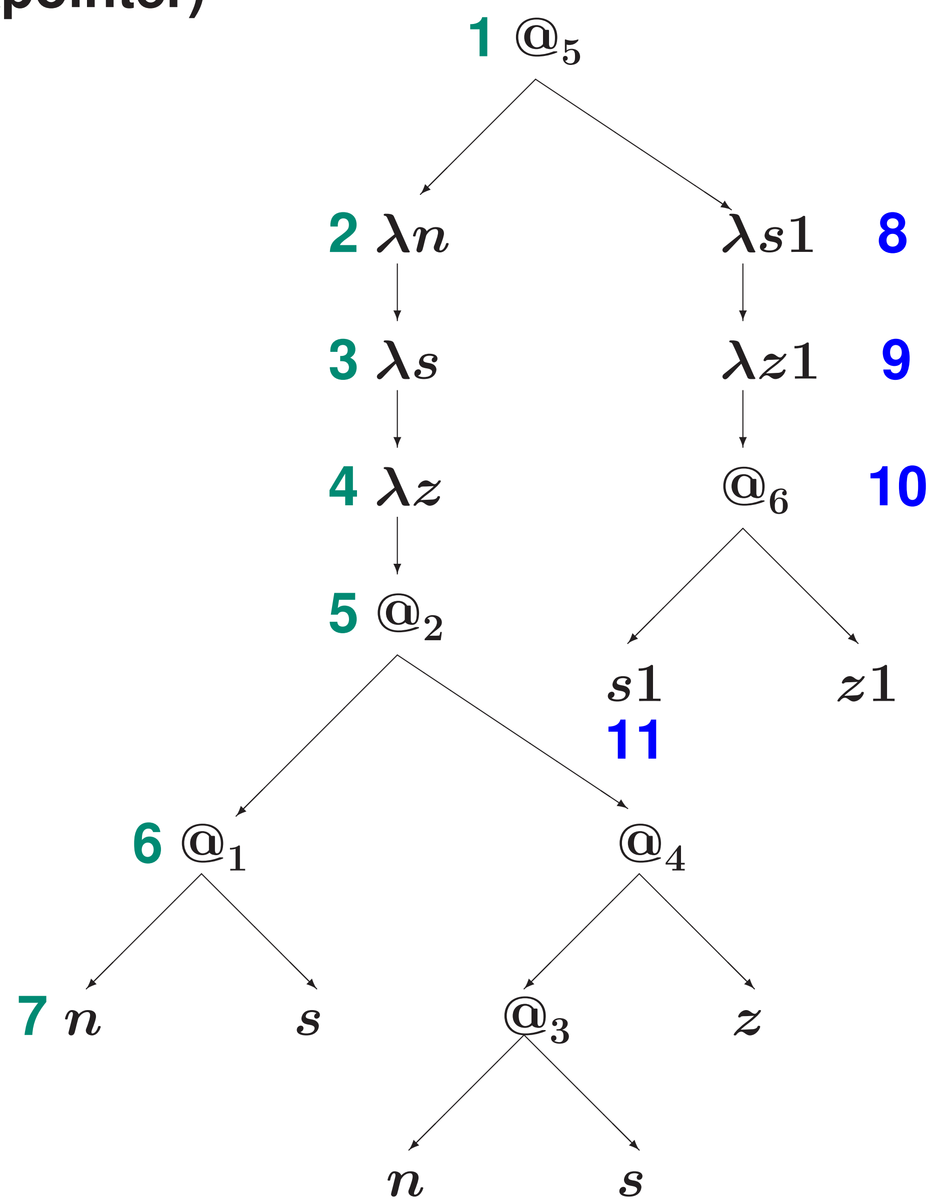
TRAVERSAL OF DOUBLE 1: STEPS 1 $\to$ 7

Save backpointers on the way down.
(backpointer: from "here" to earlier token)



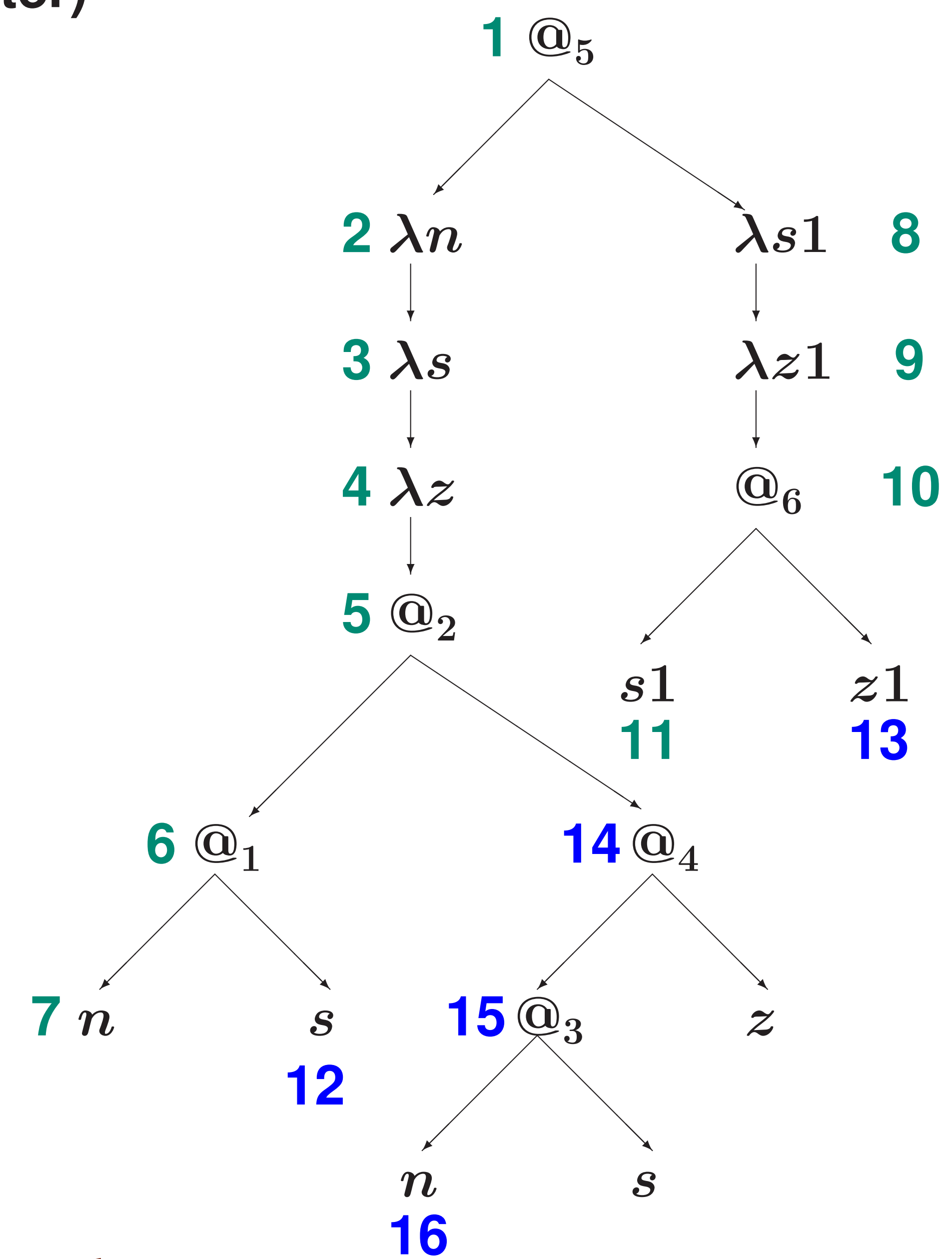
TRAVERSAL OF DOUBLE 1: STEPS 8 $\rightarrow$ 11

(find n binding to $\lambda s1\ldots$ by backpointer)



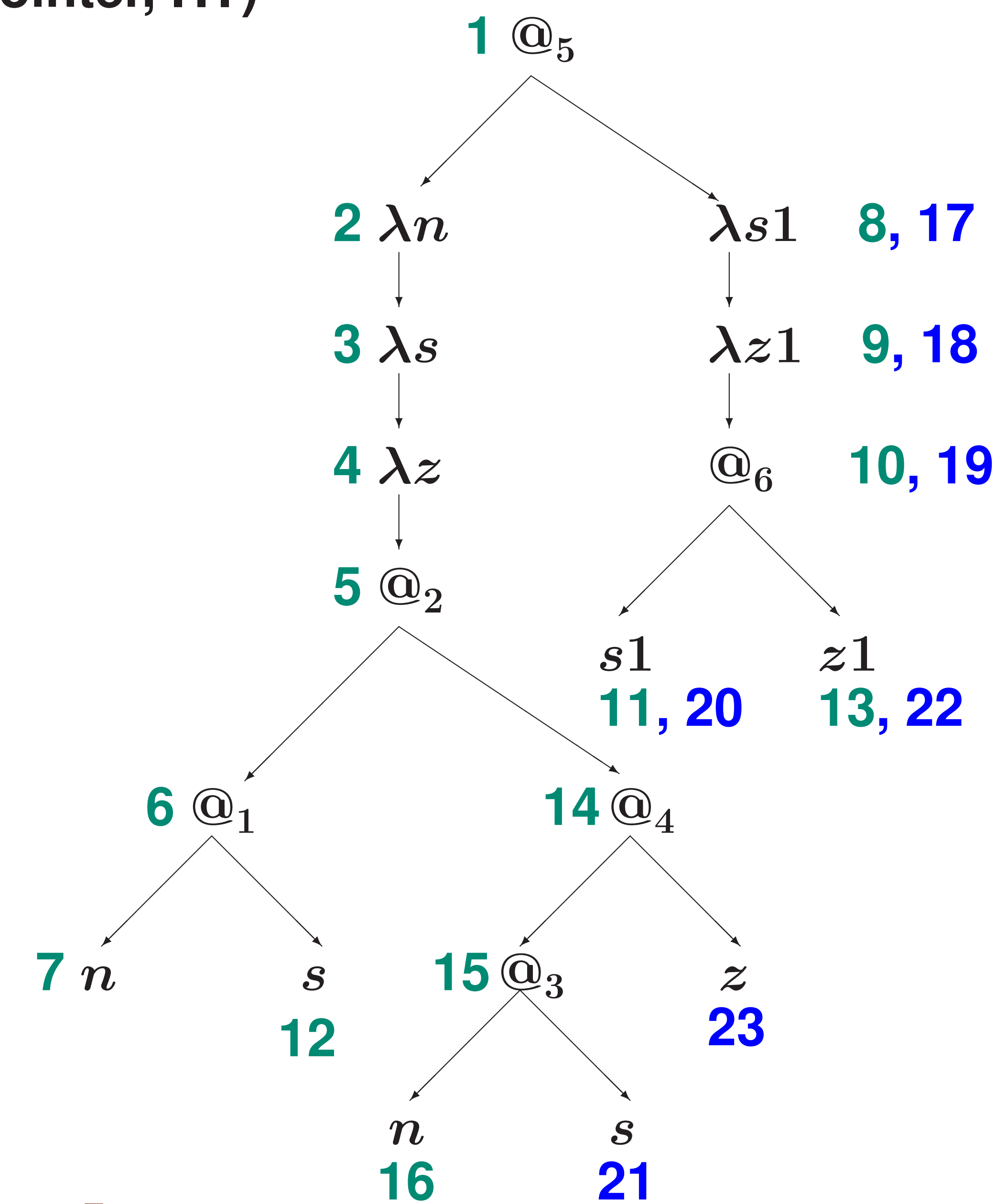
TRAVERSAL OF DOUBLE 1: STEPS 12 $\rightarrow$ 16

(find $s1$ binding to s by backpointer)



TRAVERSAL OF DOUBLE 1: STEPS 17 $\to$ 23

(find second n binding by backpointer, . . .)



OVERVIEW

A view of the Oxford normalisation procedure (ONP for short): It is

an interpreter for λ-expressions

- ONP systematically builds traversal set $\mathfrak{Trav}(M)$. What and How?
- Traversal : $tr = t_0 \cdot \ldots \cdot t_n$ t_i is a **token** (subexpression of M)
- **Syntax-directed inference rules**: based on syntax of the end-token t_n
- 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 none; depends on form of token t_n)

Data types:

$tr \in Tr = Item^*$ Traversal = a list of items

$Item = subexpression(M) \times Tr$ Item = a token and a back pointer

SOME CHARACTERISTICS

Oxford 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: by game semantics and categories, using M's types

Properties of the normalisation procedure:

ONP uses no β-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 λ-calculus into a low-level language.

PARTIAL EVALUATION, BRIEFLY

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

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

- 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*** $p_s \stackrel{def}{=} [\![spec]\!](p, s)$.
- 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.
- Net effect: a ***staging transformation***: $[\![p]\!](s, d)$ is a **1 stage** computation; but $[\![[\![spec]\!](p, s)]\!](d)$ 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.

COULD NORMALISATION BE STAGED?

1. The $spec$ equation for a normaliser program NP:

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

2. λ-calculus tradition: M is self-contained; there is no dynamic data.

   So why break normalisation into 2 stages?

   (a) The specialiser output $\textsf{NP}_M = [\![spec]\!](\textsf{NP}, M)$ can be in a much simpler language than the λ-calculus.

   Our candidate: LLL, a "low-level language" (syntax later).

   (b) A next step: consider the computational complexity of normalising, if M is applied to an external input d at run-time.

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

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

THE RESIDUAL PROGRAM NP$_M = [\![spec]\!]$ 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
- So NP$_M$ performs no operations at all on lambda expressions (!)
- NP$_M$ contains "residual code":
  - operations to extend the traversal; and (sometimes)
  - operations to follow back pointers
- Subexpressions of M will appear, but are only used as **tokens**:

  Tokens are indivisible, only used for equality comparisons with other tokens

THE LOW-LEVEL LANGUAGE LLL

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 Programming Perspective

SYNTAX

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

e       ::=  x     | f e                                          call in tail position
          |  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:

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

A token, or a product type, has a static structure, fixed for any one program. A list type `[tau]` **is dynamic, with constructors** `[] and :`

HOW TO PARTIALLY EVALUATE NP (IN PROGRAM FORM) WITH RESPECT TO STATIC λ-EXPRESSION M ?

1. Annotate parts of NP as either static or dynamic. Variables ranging over

   (a) tokens are static, i.e., λ-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 later)

   - Perform fully static computations at partial evauation time.
   - Operations to build or test a traversal: generate residual code.

STATUS: OUR WORK ON SIMPLY-TYPED λ-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 program size is only linearly larger than M, satisfying
$$|p_M| = O(|M|)$$
5. We have handwritten ONP-gen: a *generating extension* of ONP. Symbolically,
$$\text{If } p_M = [\![\text{ONP-gen}]\!]^{scheme}(M) \text{ then } \forall M \;.\; [\![M]\!]^{\Lambda} = [\![p_M]\!]^{LLL}$$
Currently: LLL = `scheme`, so the output p_M is a SCHEME program.
6. Next step: make LLL a clean stand-alone subset of HASKELL

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. Property of UNMIX: the generating extension's output programs are 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 first-order language with HASKELL-like syntax.

5. Then produce programs in LLL instead of SCHEME.

STATUS: OUR WORK ON THE UNTYPED λ-calculus

1. UNP is a normaliser for $\Lambda^{untyped}$.
2. A single traversal item may have two back pointers (in comparison: ONP uses 1).
3. UNP is defined semi-compositionally by recursion on syntax of λ-expression M.
4. UNP has been done in HASKELL and works on a variety of examples. (A more abstract definition of UNP is on the way.)
5. By specialising UNP, an arbitrary untyped λ-expression can be translated to LLL.
6. Correctness proof: pending.
7. No SCHEME version or generating extension has yet been done.

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. An example: `mul`, usual λ-calculus definition on Church numerals.

3. Loops from out of nowhere:

   - Neither `mul` nor the data contain loops;
   - but `mul` is compiled into an LLL-program with two nested loops. Applied to two Church numerals, it computes their product.
   - 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. Think about complexity and data-flow analysis of such programs.

SOME RELATED WORK

SCHEME EXAMPLE: ONP SPECIALISED TO M = `mul 3 2`

$M = \lambda m\, n\, s\, z\,.\, m(n\, s) z$ **multiplies two Church numerals.**

η-long form: longish!

Residual code to add traversal items.

- **Tokens**: $A, B, \ldots \in Subexpressions(M)$
- **Functions** $a, b, \ldots : Tr \to Tr$ (**big-step:**$current\ trav \to final\ trav$)

```
main    =  print  (reverse (a [ ⟨A []⟩ ]))  -- outermost λ()

a tr = b (⟨B tr⟩ : tr)                      -- long apply @

b tr = c (⟨C tr⟩ : tr)                      -- control  to λmnsz.m(ns)z

c tr = d (⟨D (dynamicbinder C tr)⟩ : tr)   -- find binder of variable m
d tr = cgoto1 tr                           -- control transfer to m's value

e tr = f (⟨F (dynamicbinder C tr)⟩ : tr)   -- find binder of variable n
f tr = cgoto2 tr                           -- control transfer to n's value
g tr = h (⟨H (dynamicbinder C tr)⟩ : tr) -- λ.e: find s binder
h tr = cgoto3 tr                         -- control transfer to s's value
-- etc, one for each M subexpression
```

VARIABLES: BACKPOINTER SEARCH

Suppose a variable $\mathtt{x}_i$ is encountered, and it is bound **statically** by abstraction node $\lambda\, x_1 \ldots x_n\, .\, N$ in λ-expression M.

Function `dynamicbinder` is given

`have` = a static abstraction node $\lambda\, x_1 \ldots x_n\, .\, N$
`tr` = the current traversal

It will follow back pointers in the current traversal, to find

- the **dynamic** token in the traversal
- that contains this **static** binding of variable $\mathtt{x}_i$.

```
dynamicbinder have tr =
 case tr of
 ⟨r tr'⟩ : tr'' ->
   if r == have
   then tr
   else case tr' of
        ( _ : tr''') ->  dynamicbinder have  tr'''   (follow the back pointer)
        [ ]          -> - BUG -
```

“COMPUTED GOTO” FUNCTION FOR x_i IN $\lambda x_1 \cdots x_n \,.\, N$

The idea:

Function `cgoto`$\boldsymbol{i}$`(tr)` **realises a control transfer to the item** $\langle token, bp\rangle$ **in traversal** `tr` **for the value of** x_i **in** $\lambda x_1 \cdots x_n \,.\, N$**.**

```
cgoto1 ⟨C _⟩ : tr = oa ⟨OA tr⟩ : tr
cgoto1 ⟨E _⟩ : tr = ob ⟨F tr⟩ : tr
cgoto1 ⟨G _⟩ : tr = ac ⟨H tr⟩ : tr
cgoto1 ⟨I _⟩ : tr =  i ⟨I tr⟩ : tr
cgoto1      _      = ⟨BUG []⟩ : tr

cgoto2 ⟨C _⟩ : tr = wa ⟨WA tr⟩ : tr
cgoto2 ⟨E _⟩ : tr =  m ⟨M tr⟩ : tr
cgoto2 ⟨G _⟩ : tr =  k ⟨K tr⟩ : tr
cgoto2      _     =  ⟨BUG []⟩ : tr

cgoto3 ⟨C _⟩ : tr = ac ⟨AC tr⟩ : tr
cgoto13     _     = ⟨BUG []⟩ : tr

cgoto4⟨C _⟩ : tr  = ag ⟨AG tr⟩ : tr
cgoto4      _     = ⟨BUG []⟩ : tr
```

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 without variable environments, thunks, etc:

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

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

A PARTIAL EVALUATOR COMPILES FROM Λ TO LLL

Given a traversal algorithm NP and a λ-expression M, the partial evaluato yields an LLL program. The net effect is to factor:

$$\text{NP} : \Lambda \rightarrow \mathit{Traversals}$$

into two stages:

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

where

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

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

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