Good afternoon, my name is Daniil and I'm going to present our joint with professor Neil Deaton Jones work in progress called "Partial Evaluation and Normalization by traversals".

==
=== slide ??: introduction ================================
==

The game semantics for PCF can be thought of as a PCF interpreter. In a set of game semantics papers the denotation of an expression is a game strategy. When played, the game results in a traversal.

Ong's recent paper [14] shows that a *simply-typed lambda-expression* M can be evaluated (i.e. normalised) by the algorithm that constructs a traversal of M.

A *traversal* in this case is justified sequence of subexpressions of M.
For brivety, we will call them *tokens*. One can think that M is a program and token is a program point. Any token may have a *back pointer*, aki justifier, to some other token in history. This pointers are used to lookup some information about λ-expressionin history and to find a binamic binder for variables.

Suprisely, this approach to normalization can be implemented with *none of the traditional implementation technics* like β-reductions, environments that binds variables to values, closures and thunks for function calls and parameters.

==
=== slide ??: start point ================================
==

Now then, it looks very promising to study an *operational* consequences of game semantics. And a start point for our research is an Oxford normalization procedure. We will use an abbriviation ONP for name it.

ONP can be thought as a an interpreter for λ-expressions. It constracts a set of traversals $\mathfrak{Trav}(M)$ for a given λ-expression$M$ each of that is a <u>path</u> in a tree that represents a normal form of λ-expression M.

But what and how?

Consider a traversal tr from $\mathfrak{Trav}(M)$ such that is a sequence from t_0 to t_n where each t_i is a token.

On each step ONP for all such traversals from $\mathfrak{Trav}(M)$ adds one or more externsions in $\mathfrak{Trav}(M)$ or maybe zero. If zero then it means that traversal tr is already a path in a tree representing an β-normal η-long form of the term.

Each extension is a traversal obtained by contatination traversal tr with <u>one new token</u> t' that could have a backpointer somewhere in a current view.

Moreover, ONP uses a inverence rules that are entirely based on a sintax of the end-token t_n. In other words, ONP is based on syntax-directed inference rules.

Thus, we have two main <u>data types</u>: traversal and items. <u>Traversal</u> is just a list of <u>items</u> where an item is a pair: a token and a backpointer.

==
=== slide ??: SOME ONP CHARACTERISTICS ===============
==

Now then, let's summarize some importants properties of Oxford normaliztion procedure.

Fisrt, ONP can be abblied only to simply-tiped λ-terms that are in a η-long form.

The long form is obtained by η-expanding term fully and replacing the implicit binary application operator of each redex by the *long application operator* @.

Second, *ONP* realises the complete head linear reduction. Complete head linear reduction is a repeated applying of *head linear reduction* to all arguments while head linear reduction itself on each step substitutes only one occurence of variable that is a *hoc-position*.

The correctness of normalization by traversals is done by game semantics and categories, and fulle based on M's types.

Then, by contract to standart approaches to term normalization, method of normalization by traversals uses no β-reduction and leavs the original term inpact.

Finally, an important property of *ONP* is that while running *ONP* does not use the types od M at all. As was mention before, *ONP* constracts straversal using only syntax-derected rules.

And now now we are able to formulate **goals** of this research:

- First goal is to extend ONP to UNP that is a normalizer for the *untyped* lambda calculus and
- Second, Partially evaluate a normaliser with respect to "static" input λ-term M. By partial evaluating normalizer it becoms possible to compile λ-calculus into a *low-level language*(*LLL* for brivety) that futher can be used to express semantics.

==
=== slide ??: PARTIAL EVALUATION, BRIEFLY =========
==

Now I whant breafly remined what is partial evaluation and how it can be applied to normalizer.

A partial evaluator is a program specialiser that can be defined as follows: $spec$:
$\forall p \in Programs\ .\ \forall s,\ d \in Data\ .\ [\![[\![spec]\!](p, s)]\!](d)\ =\ [\![p]\!](s, d)$
Specializer is a program $spec$ that being applied to initial program p and data s and then to the remaining data d is absolutely the same that apply initial program p to both data s and d.

- In other words, this means that for a given program p and some static data s, partial evaluator builds a residual program $p_s \stackrel{def}{=} [\![spec]\!]\ p\ s$ that being applied to the remaing data d produces the same result as an initial program p being applied to both of them.
- A net effect of partial evaluation is a staging program transformation. I.e. partial evaluation devides computation in two stages: optimized residual program generation and running this program on some dunamic data. While running an initial program p on data s and d is a one stage computation.
- Partial ervaluation produce some speed up by precomputing all static input at compile time. It can be applied in compilation, moreover PE permits an automatic compiler generation from an interpreter by self-applying a specializer.

==
=== slide ??: why partially evaluate ONP =============
==

The $spec$ equation for a normaliser program **NP** that is just a function from lambda-calculus to Traversals can be seen on the slide.

You can see that there is no external dynamic data and that is a tradition for λ-calculus that λ-term M is self-contained.

So a question is **why break normalization into two stages?**

Well, ... , there are several reasons for it:

1. First, a specialiser output $\mathrm{NP}_M = [\![spec]\!](\mathrm{NP}, M)$ can be in a much simplier language than λ-calculus. And our candidate for it is some **low-level language** called LLL.

2. Second, two stages will be natural for *semantics-directed compiler generation*.
   Our aim is to use LLL as an intermediate language to express semantics.
   Programs on this low-level language can be thought as a semantics for programs from λ-calculus.
   In other words, we factor the initial normalization procedure NP into two stages:
   First stage that called NP_1 is a result of partially evaluating normalization procedure to input term M
   $\mathrm{NP}_1 \ = \ [\![spec]\!] \ \mathrm{NP} \ M \ : \ \Lambda \to LLL$
   and the second stage, NP_2 is the *semantic function* of LLL-programs
   $\mathrm{NP}_2 \ = \ [\![\cdot]\!] \ : \ LLL \to Traversals.$

==
=== slide ??: how to partially evaluate ONP ==========
==

A next question is "how to partially evaluate oxford nomalization procedure with respect to the static input term M"?

1. First, we have to annotate parts of normalization procedure as either static or dynamic. And here variables ranging over

   (a) tokens that are static,
   Because there are only finitely many subexpressions of M.

   And all other data is actually dynamic

   (b) i.e. back pointers are dynamic;

   (c) and so the traversal being built from both of them is dynamic too.

2. Computations in normalization proicedure **NP** are either unfolded by partial evaluator in compile time or residualised that means that partial evaluator will generate a runtime code to do them later in run-time.
   And then

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

==
=== slide ??: The residual program $ONP_M = [\![spec]\!]$ NP M ===============
==

Now we will talk about structure of a spesialized program ONP_M.

Note, that ONP is not quite structually inductive but it is semi-compositional: in a sence that Any recursive ONP call has a substructure of M as argument.

This property has several consequences:

- Firts, the partial evaluator can do, at specialisation time, all of the ONP operations that depend only on input term M
- wherein this also means that ONP_M performs no operations at all on lambda expressions(!)

  and for all other operations

- the specialised program ONP_M will be generated. This program contains "residual code", that means that it contains only operations to build the traversal. There are two kind of operation to do this:
  - operations to extend the traversal; and sometimes
  - operations to follow back pointers when needed to do this
- An important fact is that subexpressions of M will appear, but are only used as tokens:
  This means that tokens are indivisible: they are only used as labels (i.e. program points) and for equality comparisons with other tokens. Actually we use names instead of real subexpressions. And real subexpressions is only needed for the normalized term reconstruction from traversals.

==
=== slide ??: Status: our work on simply-typed λ-calculus===
==

Status of our work for symply-tiped λ-calculus is as follows:

1. We have one version of ONP writen in Haskell and another in Scheme
2. The Haskell version includes: typing using algorithm W; conversion to eta-long form; the traversals generation algorithm itself; and construction of the normalised term from the set of traversals.
3. While Scheme version is nearly ready to apply automatic partial evaluation. We are planning to use the unmix partial evaluator that is written by Sergei Romanenko and others. unmix is a general partial evaluator for Scheme. We expect that we will achieve the described above effect of specialising ONP.
4. An important fact is that the LLL output program size is only linearly larger than M, satisfying
$$|p_M| = O(|M|)$$
   while traversal itself can be unboundaly larger than size of the input term.

5. We have also have a handwritten a *generating extension* of *ONP* $ONP - gen$. Symbolically,

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

   It means that by given λ-term M $OMP - gen$ generates an LLL program p_M that being executed generates a traversal for M.

   For now: *LLL* is a tiny subset of `scheme`, so the output p_M is a scheme program.

6. There are more thing to do for simply-typed λ-calculus:

   - First, produce a generating extension automatically by specialising the specialiser to a Λ-traverser using unmix.
   - Second, redefine *LLL* formally as a *clean stand-alone tiny first-order subset* of haskell and use haskell supersompiler to achive a partial evaluation effect.
   - Also we whant to extend existing approach to programs with input dynamic data in run-time.

===
=== Status: our work on the untyped λ-calculus======
===

For untyped λ-calculus

1. We have a normaliser *UNP* that is a normaliser for arbitrary untyped lambda term.

2. *UNP* has been done in Haskell and works on a variety of examples. Right now we are working on a more abstract definition of *UNP*.

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 λ-expression M. This actually means that it also can be specialized as *ONP* can.

5. Moreover, by specialising *UNP*, an arbitrary untyped λ-expression can be translated to our low-level language. So the specialised version of *UNP* could be a semantic function for λ-calculus.

6. Correctness proof: pending. For now, we are working on a correctness proof for UNP. We expect that we will prove its correctness formally using some proof assistant like **Coq**.

===
=== Status: Towards separating programs from data in Λ ==========================

Also we have a one more direction for research:

1. An idea is to 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 intresting examples in this case a is usual λ-calculus definition of multiply function ($mult$) on Church numerals.

3. Amaizing fact is that **Loops from out of nowhere**:

   - Neither `mult` nor the data contain loops;
   - but `mult` function is compiled into an *LLL*-program with two nested loops, one to each input numeral, that being applied to two Church numerals computes their product.
   - We 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*.