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

1. Introduction

#TODO: Reformulate: game semantics for various versions of lambdas.
#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.

For example, Luke Ong's recent paper "Normalization by Traversals" shows, that a *simply-typed lambda-expression* M can be evaluated (i.e. normalized) by the algorithm that constructs a traversal of M.

A *traversal* in this case is a justified sequence of subexpressions of M. For the sake of brevity, we will call them *tokens*. One may think, that M is a program and token is a program point. Any token may have a *back pointer*, or justifier, to some other previously encountered token. These pointers are used to lookup some information about λ-expressionin the history of computation and to find dynamic binders for variables.

Hereafter we will call this approach to normalization of simply typed lambda terms Oxford Normalization Procedure, or ONP. For a given λ-expression$M$ *ONP* constructs a set of traversals $\mathfrak{Trav}(M)$, each of which corresponds to some path in normalized term tree.

2. Starting point

Confirm the understanding of operational view and its motivation.
In the context of our research, we are interested in *operational*, algorithmic view of game semantics-based normalization without direct reference to game semantic foundations (?).

Consider a traversal tr from $\mathfrak{Trav}(M)$, which is a sequence of tokens t_0 t_n.

For all such traversals from $\mathfrak{Trav}(M)$ *ONP* on each step adds zero or more extensions to $\mathfrak{Trav}(M)$. If none is added, then the traversal tr already represents some path in the tree of β-normal η-long form of the term.

Each extension is a traversal, obtained by concatenation of traversal tr and one new token t', which may be equipped with a pointer to some previous token in tr.

Moreover, *ONP* uses inference rules, discriminated solely by sintax of end-tokens t_n. In other words, *ONP* is syntax-directed.

In terms of implementation, we have two main data types: traversals and items. Traversal is just a list of items, where an item is a pair: a token and its (optional) backpointer.

3. SOME ONP CHARACTERISTICS

Now then, let's summarize some importants for us properties of ONP.

Fisrt, *ONP* can be applied only to simply-tiped λ-terms in η-long form. The η-long form is obtained by full η- expansion of the term and substitution of all implicit binary application operators for each redex by *long application operator* @. The crucial point is that in order to perform this expansion one needs to know the exact types of all subterms.

Second, *ONP* implements complete head linear reduction. Complete head linear reduction can be seen as regular *head linear reduction* followed by regular *head linear reductions* in all arguments.

The correctness of normalization by traversals is proven in terms of game semantics and categories, and fully based on M's types.

Then, in contrast to standart evaluation approaches, normalization by traversals uses no β-reductions, leaves the original term intact, and can be implemented without *traditional techniques* like environments, closures etc.

Finally, an important property of *ONP* is that, while running, *ONP* does not use type information at all. As it was mentioned before, *ONP* constructs traversals using only syntax-derected rules.

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

- The first goal is to extend ONP to the untyped case (hereafter UNP);
- The second is to reexamine the outcomes of partial evaluation in the light of alternative evaluation technique.

4. PARTIAL EVALUATION, BRIEFLY

Now let us briefly recollect, what is partial evaluation.

Partial evaluation is a program optimization technique, also known as program specialization. A one-argument function can be obtained from function of two arguments by specialization, i.e. by "freezing" one of its inputs to a fixed value. A partial evaluator for a given subject program together with a part of its input data s constructs a new program p_s, which, given p's remaining input d, yields the same result as p would produce for both inputs.

- In the context of partial evaluation, data s is called static, data d — dynamic, and program p_s is called residual, or specialized.
- Partial evaluation yeilds program speed up by precomputing all static input at compile time.
- A net effect of partial evaluation is a staging program transformation. Partial evaluation divides one-stage program computation in two stages:
  1. optimized residual program generation and;
  2. execution of residual program for some dynamic input.
- Most successful applications of PE are compilation and compiler generation.

  For example, specialization of an interpreter with respect to a source program yields a target program. Moreover, a well-known fact about partial evaluation is that if partial evaluator is self-applicable, then compiler generation is possible by specializing the partial evaluator itself on a fixed interpreter yields a compiler. Finally, specializing a partial evaluator on itself yields a compiler generator.
- An old idea: use partial evaluation to provide Semantics-directed compiler generation. By this we mean more than just a tool to help humans write compilers. Given a specfication of a programming language, like a formal semantics or an interpreter,

the goal is to automatically and correctly transform it into a compiler. The motivation for automatic compiler generation is evident: the three jobs of writing the language specfication, writing the compiler, and showing the compiler to be correct are reduced to one: writing the language specfication in a form suitable as input to the compiler generator.

5. Why partially evaluate NP?

An instance of $spec$ equation for a normalizer program **NP**, which is just a function from λ-calculus to Traversals is shown on the slide.

You can see that there are no external dynamic data and, conventionally, λ-term M is self-contained.

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

Well, $\ldots$, there are several reasons:

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

2. Second, two stages are natural for *semantics-directed compiler generation*.

   Our aim is to use *LLL* as an intermediate language to express semantics. This means, that programs in this low-level language can be thought as a semantics for programs in λ-calculus. In other words, we factor the initial normalization procedure NP into two stages:

   The first stage, denoted NP_1, is a result of specialization of normalization procedure on the input term M

   $$\mathrm{NP}_1 \;=\; [\![spec]\!]\ \mathrm{NP}\ M \;:\; \Lambda \to LLL$$

   and the second stage, NP_2, is *semantic function* for *LLL*-programs:

   $$\mathrm{NP}_2 \;=\; [\![\,\cdot\,]\!] \;:\; LLL \to Traversals$$

6. How to partially evaluate *ONP*?

The next question is ”how to partially evaluate oxford nomalization procedure with respect to a static input term M”?

A well-known fact is that partial evaluation would achieve the best result if the input program is *annotated*. Thus,

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

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

   (b) back pointers are dynamic;

   (c) traversals being built from both of them are dynamic too.

2. Then, we can run partial evaluator on annotated program. Computations in normalization procedure *ONP* are either unfolded by partial evaluator at compile time or residualised, which means, that partial evaluator generates some runtime code to execute them later at run- time.

   Finaly,

   - Perform all fully static computations at partial evauation time.
   - Generate residual code for all other operations.

7. The residual program $ONP_M = [\![spec]\!]$ NP M

Now let's discuss the structure of specialized program ONP_M.

Note, that *ONP* is not quite structually inductive, but semi- compositional in the sense, that any recursive *ONP*-call has a substructure of M as argument.

This property has several consequences:

- First, the partial evaluator can do, at specialization time, all of the *ONP* operations, which depend only on input term M
- wherein this also means, that ONP_M performs no operations at all on lambda expressions

For all other operations

- a specialised program ONP_M will be generated. This program will contain "residual code", containing only operations to build the traversal. There are two kind of operations:
  - 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 will be only used as tokens:
  This means that tokens are indivisible: they are only used as labels (i.e. program points) and for equality comparison with other tokens. Actually we use names instead of real subexpressions. And real subexpressions are only needed for the normalized term reconstruction from traversals.

8. Status: our work on simply-typed λ-calculus

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

1. We have one version of *ONP*, written in Haskell, and another in Scheme
2. The Haskell version includes: typing using algorithm W; conversion to eta-long form; the traversal generation algorithm itself; reconstruction of the normalized term from the set of traversals.
3. The Scheme version is nearly ready to apply automatic partial evaluation. We are planning to use the unmix partial evaluator, which was 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 size of output LLL program is only linearly larger,than M, while traversal itself can be arbitrarily larger, than the size of the input term.

5. We also have a handwritten *generating extension* for ONP $ONP - gen$. It means, that for given λ-term M $OMP - gen$ generates an LLL program p_M, which, 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 supercompiler to achive the effect of partial evaluation.
   - Also we want to extend existing approach to programs with input dynamic data at run-time.

9. Status: our work on the untyped λ-calculus

For untyped λ-calculus

1. We have a normaliser UNP, which works for arbitrary (normalizing) untyped lambda terms.

2. UNP is implemented in Haskell and works for a variety of examples. Right now we are working on a more justified definition of UNP.

3. Some of traversal items may have two back pointers; for comparison: ONP uses only one.

4. As ONP, UNP is also defined semi-compositionally via recursion on syntax of λ-expression M. This actually means that it also can be specialized similarly to ONP.

5. Moreover, by specializing 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. For now, we are working on a correctness proof for $\boldsymbol{UNP}$. We expect that we will prove its correctness formally using some proof assistants like **Coq**.

10. Towards separating programs from data in Λ

Also we have a one more direction for research:

1. An idea is to consider a computation of λ-expression M on input d as a two-player game between LLL-programs for M and d.

2. An intresting example here is the conventional λ-calculus definition of multiply function ($mult$) on Church numerals.

3. Amasing fact is that **Loops come 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 for each input numerals. This function, when applied to two Church numerals, computes their product.
- We expect, that in this perticular case we perform the computations entirely without back pointers.

4. Right now we are trying to express such program-data games in a *communicating* version of *LLL*.