

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".

==== slide ??: 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. normalised) by the algorithm that constructs a *traversal* of M .

A *traversal* in this case is a justified sequence of subexpressions of M . For brevity, 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*, or justifier, to some other previously encountered token. These pointers are used to lookup some information about λ -expression in 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 $\mathcal{T}rav(M)$, each of which corresponds to some *path* in normalized term tree.

==== slide ??: 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 $\mathcal{T}rav(M)$, which is a sequence of tokens $t_0 t_n$.

For all such traversals from $\mathcal{T}rav(M)$ *ONP* adds on each step zero or more extensions to $\mathcal{T}rav(M)$. 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 syntax 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.

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

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

First, *ONP* can be applied only to *simply-typed* λ -terms in *η -long form*. The η -long form is obtained by η -expanding term fully and replacing implicit binary application operator of each redex by the *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* of 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 standard evaluation approaches, normalization by traversals uses **no β -reductions**, leaves the original term intact, and can be implemented without resorting to *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-directed 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.

=====

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

=====

Now let briefly remind what is partial evaluation and discuss an effect of its application to normalizer.

Partial evaluation is a program optimization technique also known as **program specialization**. A one-argument function can be obtained from one with two arguments by specialization, i.e. by “freezing” one input to a fixed value. A partial evaluator by given subject program together with part of its input data s constructs a new program p_s which, when given p ’s remaining input d , will yield the same result that p would have produced given both inputs.

- Partial evaluation yields program speed up by **precomputing** all static input at compile time.
- Thus, while talking about partial evaluation data s is called **static**, data d — dynamic, and program p_s that is build by specializer is called **residual**.
- 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. running the generated program on some dynamic data.
- Most successful applications of PE are **compilation** and **compiler generator**. For example, partial evaluation of an interpreter with respect to a source program yields a target program. Moreover, a well-known fact about partial evaluation is that if provided the partial evaluator is self-applicable, then **compiler generation** is possible by specializing the partial evaluator itself with respect to a fixed interpreter yields a compiler. Finally, specializing the partial evaluator with respect to 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 specification of a programming language, like a formal semantics or an interpreter, the goal is automatically and correctly transform it into a compiler. The motivation for automatic compiler generation is evident: the three jobs of writing the language specification, writing the compiler, and showing the compiler to be correct are reduced to one: writing the language specification in a form suitable as input to the compiler generator.

=====
 === 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 $NP_M = \llbracket spec \rrbracket (NP, M)$ can be in a **much simpler 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*
 $NP_1 = \llbracket spec \rrbracket NP\ M : \Lambda \rightarrow LLL$
 and the second stage, NP_2 is the *semantic function* of LLL-programs
 $NP_2 = \llbracket \cdot \rrbracket : LLL \rightarrow Traversals$.

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

A next question is "how to partially evaluate oxford normalization 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 procedure **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 evaluation time**.
 - and for operations to build or test a traversal: generate **residual code**.

==== slide ??: The residual program $ONP_M = \llbracket spec \rrbracket NP\ M$ =====

Now we will talk about structure of a specialized 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 written 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,

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

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 interesting example in this case is a usual λ -calculus definition of multiply function (*mult*) on Church numerals.
3. Amazing 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*.