Working Notes: Compiling ULC to Lower-level Code by Game Semantics and Partial Evaluation

Daniil Berezun[1] and Neil D. Jones[2]

[1] JetBrains and St. Petersburg State University (Russia)
[2] DIKU, University of Copenhagen (Denmark)

Abstract. What: Any expression M in ULC (the untyped λ-calculus) can be compiled into a rather low-level language we call LLL, whose programs contain *none of the traditional implementation devices for functional languages:* environments, thunks, closures, etc. A compiled program is first-order functional and has a fixed set of working variables, whose number is independent of M. The generated LLL code in effect *traverses* the subexpressions of M.
How: We apply the techniques of *game semantics* to the untyped λ-calculus, but take a more operational viewpoint that uses much less mathematical machinery than traditional presentations of game semantics. Further, the untyped lambda calculus ULC is compiled into LLL by *partially evaluating* a traversal algorithm for ULC.

1 Context and contribution

Plotkin posed the problem of existence of a fully abstract semantics of PCF [17]. Game semantics provided the first solution [1–3, 9]. Subsequent papers devise fully abstract game semantics for a wide and interesting spectrum of programming languages, and further develop the field in several directions.

A surprising consequence: it is possible to build a lambda calculus interpreter with **none** of the traditional implementation machinery: β-reduction; environments binding variables to values; and "closures" and "thunks" for function calls and parameters. This new viewpoint on game semantics looks promising to see its operational consequences. Further, it may give a new line of attack on an old topic: *semantics-directed compiler generation* [10, 18].

Basis: Our starting point was Ong's approach to normalisation of the simply typed λ-calculus(henceforth called STLC). Paper [16] adapts the game semantics framework to yield an STLC normalisation procedure (STNP for short) and its correctness proof using the traversal concept from [14, 15].

STNP can be seen as in *interpreter*; it evaluates a given λ-expression M by managing a list of subexpressions of M, some with a single back pointer. These notes extend the normalisation-by-traversals approach to the *untyped* λ-calculus, giving a new algorithm called UNP, for Untyped Normalisation Procedure. UNP correctly evaluates any STLC expression sans types, so it properly extends STNP since ULC is Turing-complete while STLC is not.

Plan: in these notes we start by describing a weak normalisation procedure. A traversal-based algorithm is developed in a systematic, semantics-directed way. Next step: extend this to full normalisation and its correctness proof (details omitted from these notes). Finally, we explain briefly how partial evaluation can be used to implement ULC, compiling it to a low-level language.

2 Normalisation by traversal: an example

Perhaps surprisingly, the normal form of an STLC λ-expression M may be found by simply taking a walk over the subexpressions of M. As seen in [14–16] there is no need for β-reduction, nor for traditional implementation techniques such as environments, thunks, closures, etc. The "walk" is a *traversal*: a sequential visit to subexpressions of M. (Some may be visited more than once, and some not at all.)

A classical example; multiplication of Church numerals[3]

$$\texttt{mul} = \lambda m.\lambda n.\lambda s.\lambda z.m(ns)z$$

A difference between reduction strategies: consider computing $3*2$ by evaluating $\texttt{mul}\ \underline{3}\ \underline{2}$. Weak normalisation reduces $\texttt{mul}\ \underline{3}\ \underline{2}$ only to $\lambda s.\lambda z.\underline{3}(\underline{2}s)z$ but does no computation under the lambda. On the other hand strong normalisation reduces it further to $\lambda s.\lambda z.s^6 z$.

A variant is to use free variables S, Z instead of the bound variables s, z, and to use $\texttt{mul}' = \lambda m.\lambda n.m(nS)Z$ instead of $\texttt{mul}$. Weak normalisation computes all the way: $\texttt{mul}'\ \underline{3}\ \underline{2}$ weakly reduces to $S^6 Z$ as desired. More generally, the Church-Turing thesis holds: a function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is partial recursive (computable) iff there is a λ-expression M such that for any $x_1, \ldots, x_n, x \in \mathbb{N}$

$$f(x_1, \ldots, x_n) = x \ \Leftrightarrow\ M(S^{x_1}Z)\ldots(S^{x_n}Z) \text{ weakly reduces to } \ S^x Z$$

The unique traversal of $\underline{3}\,(\underline{2}\,S)\,Z$ visits subexpressions of Church numeral $\underline{3}$ once. However it visits $\underline{2}$ twice, since in general $x * y$ is computed by adding y together x times. The weak normal form of $\underline{3}\,(\underline{2}\,S)\,Z$ is $S^6 Z$: the core of Church numeral $\underline{6}$.

Figure 1 shows traversal of expression $\underline{2}\,(\underline{2}\,S)\,Z$ in tree form[4]. The labels **1:, 2:** etc. are not part of the λ-expression; rather, they indicate the order in which subexpressions are traversed.

[3] The Church numeral of natural number $\boldsymbol{x}$ is $\underline{\boldsymbol{x}} = \boldsymbol{\lambda s.\lambda z.s^x z}$. Here $\boldsymbol{s^x} = \boldsymbol{s(s(\ldots s(z)\ldots))}$ with $\boldsymbol{x}$ occurrences of $\boldsymbol{s}$, where $\boldsymbol{s}$ represents "successor" and $\boldsymbol{z}$ represents "zero".

[4] Application operators $@_i$ have been made exlicit, and indexed for ease of reference. The two $\underline{2}$ subtrees are the "data'; their bound variables have been named apart to avoid confusion. The figure's "program" is the top part $\_(\_\,S)\,Z$.

Computation by traversal can be seen as a game

The traversal in Figure 1 is a game play between program $\lambda m \lambda n.(m@\ (n@S))@Z$ and the two data values (m, n each have λ-expression $\underline{2}$ as value).

Informally: two program nodes are visited in steps 1, 2; then data 1's leftmost branch is traversed from node $\lambda s1$ until variable $s1$ is encountered at step 6. Steps 7-12: data 2's leftmost branch is traversed from node $\lambda s2$ down to variable $s2$, after which the program node S is visited (intuitively, the first output is produced). Steps 13-15: the data 2 nodes $@_6$ and $s2$ are visited, and the program: it produces the second output S. Steps 16, 17: $z2$ is visited, control is finished (for now) in data 2, and control resumes in data 1.

Moving faster now: $@_4$ and the second $s1$ are visited; data 2 is scanned for a second time; and the next two output S's are produced. Control finally returns to $z1$. After this, in step 30 the program produces the final output Z.

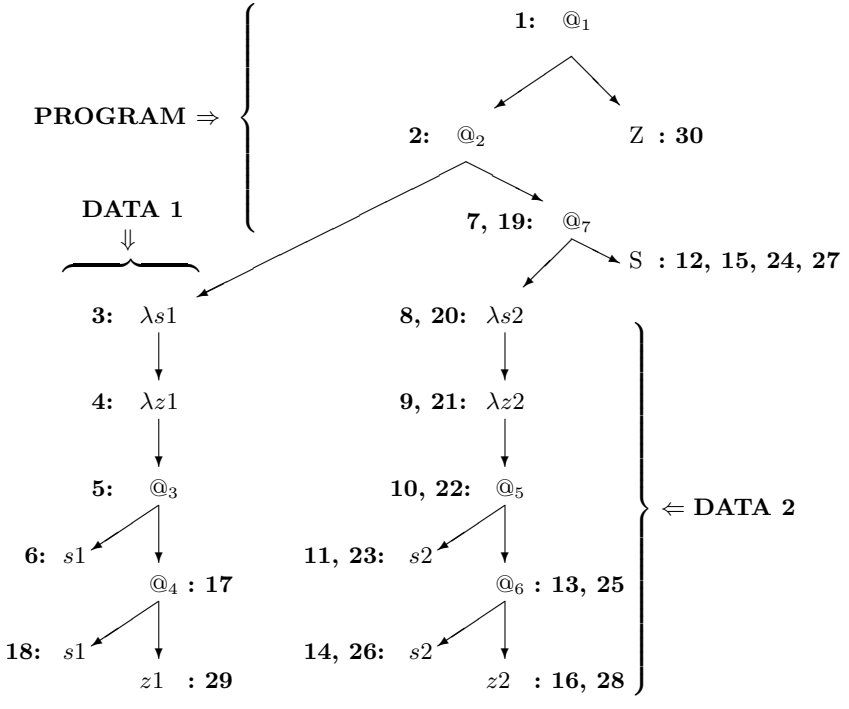


Fig. 1. Syntax tree for `mul` $\underline{2}\ \underline{2}\ S\ Z = \underline{2}(\underline{2}S)Z$. (Labels show traversal order.)

Which traversal? As yet this is only an "argument by example"; we have not yet explained *how to choose* among all possible walks through the nodes of $\underline{2}(\underline{2}S)Z$ to find the correct normal form.

3 Overview of three normalisation procedures

3.1 The STNP algorithm

The STNP algorithm in [16] is deterministic, defined by syntax-directed inference rules.

The algorithm is type-oriented even though the rules do not mention types: it requires as first step the conversion from STLC to η-long form. Further, the statement of correctness involves types in "term-in-context" judgements $\Gamma \vdash M : A$ where A is a type and Γ is a type environment.

The correctness proof involves types quite significantly, to construct program-dependent arenas, and as well a category whose objects are arenas and whose morphisms are innocent strategies over arenas.

3.2 Call-by-name weak normalisation

We develop a completely type-free normalisation procedure for ULC.[5] Semantics-based stepping stones: we start with an environment-based semantics that resembles traditional implementations of CBN (call-by-name) functional languages. The approach differs from and is simpler than [13].

The second step is to enrich this by adding a "history" argument to the evaluation function. This records the "traversal until now". The third step is to simplify the environment, replacing recursively-defined bindings by bindings from variables to positions in the history. The final step is to remove the environments altogether.

The result is a closure- and environment-free traversal-based semantics for weak normalisation.

3.3 The UNP algorithm

UNP yields full normal forms by "reduction under the lambda" using *head linear reduction* [7,8]. The full UNP algorithm [4] currently exists in two forms:

- An implementation in HASKELL; and
- A set of term rewriting rules. A formal correctness proof of UNP is nearing completion, using the theorem prover COQ [4].

[5] Remark: evaluator nontermination is allowed on an expression with no weak normal form.

4 Weak CBN evaluation by traversals

We begin with a traditional environment-based call-by-name semantics: a *reduction-free* common basis for implementing a functional language. We then eliminate the environments by three transformation steps. The net effect is to replace the environments by traversals.

4.1 Weak evaluation using environments

The object of concern is a pair $e:\rho$, where e is a λ-expression and ρ is an environment[6] that binds some of e's free variables to pairs $e'{:}\rho'$.

Evaluation judgements, metavariables and domains:

$e{:}\rho \Downarrow v$ Expression e in environment ρ evaluates to value v
$v, e{:}\rho \Downarrow v'$ Value v applied to argument e in environment ρ gives value v'
$\rho \in Env \quad = Var \rightharpoonup Exp \times Env$
$v \in Value = \{e : \rho \mid e$ does not have form $(\lambda x.e_0)e_1\}$

Note: the environment domain Env is defined recursively.

Determinism The main goal, given a λ-expression M, is to find a value v such that $M{:}[] \Downarrow v$ (if it exists). The following rules may be thought of as an *algorithm* to evaluate M since they are are deterministic. Determinism follows since the rules are *single-threaded*: consider a goal *left* $\Downarrow$ *right*. If *left* has been computed but *right* is still unknown, then at most one inference rule can be applied, so the final result value is uniquely defined (if it exists).

$$\text{(Lam)} \quad \frac{}{\lambda x.e{:}\rho \Downarrow \lambda x.e{:}\rho} \qquad \text{(Freevar)} \quad \frac{x \text{ free in } M}{x{:}\rho \Downarrow x{:}[]} \qquad \text{(Boundvar)} \quad \frac{\rho(x) \Downarrow v}{x{:}\rho \Downarrow v}$$

Abstractions and free variables evaluate to themselves. A bound variable x is accessed using call-by-name: the environment contains an unevaluated expression, which is evaluated when variable x is referenced.

$$\text{(AP)} \quad \frac{e_1{:}\rho \Downarrow v_1 \qquad v_1, e_2{:}\rho \;\Downarrow\; v}{e_1@e_2{:}\rho \;\Downarrow\; v}$$

Rule (AP) first evaluates the operator e_1 in an application $e_1@e_2$. The value v_1 of operator e_1 determines whether rule (AP_λ) or ($\text{AP}_{\overline{\lambda}}$) is applied next.

$$(\text{AP}_\lambda) \quad \frac{e' = \lambda x.e'' \qquad \rho'' = \rho'[x \mapsto e_2{:}\rho] \qquad e''{:}\rho'' \Downarrow v}{e'{:}\rho', e_2{:}\rho \;\Downarrow\; v}$$

[6] Call-by-name semantics: If ρ contains a binding $x \mapsto e' : \rho'$, then e' is an as-yet-unevaluated expression and ρ' is the environment that was current at the time when x was bound to e'.

In rule (AP_λ) if the operator value is an abstraction $\lambda x.e'':\rho'$ then ρ' is extended by binding x to the as-yet-unevaluated operand e_2 (paired with its current environment ρ). The body e'' is then evaluated in the extended ρ' environment.

$$(\mathrm{AP}_{\overline{\lambda}}) \quad \frac{e' \neq \lambda x.e'' \qquad e_2{:}\rho \Downarrow e_2'{:}\rho_2' \qquad \mathrm{fv}(e') \cap \mathrm{dom}(\rho_2') = \emptyset}{e'{:}\rho', e_2{:}\rho \;\Downarrow\; (e' @ e_2'){:}\rho_2'}$$

In rule ($\mathrm{AP}_{\overline{\lambda}}$) the operator value $e':\rho'$ is a non-abstraction. The operand e_2 is evaluated. The resulting value is an application of operator value e' to operand value (as long as no free variables are captured).

Rule ($\mathrm{AP}_{\overline{\lambda}}$) yields a value containing an application. For the multiplication example, Rule ($\mathrm{AP}_{\overline{\lambda}}$) yields all of the S applications in result $S@(S@(S@(S@Z)))$.

4.2 Environment semantics with traversal history h

Determinism implies that there exists at most one sequence of visited subexpressions for any $e{:}\rho$. We now extend the previous semantics to accumulate the history of the evaluation steps used to evaluate $e{:}\rho$.

These rules accumulate a list $h = [e_1{:}\rho_1, \ldots, , \ldots, e_n{:}\rho_n]$ of all subexpressions of λ-expression M that have been visited, together with their environments. Call such a partially completed traversal a *history*. A notation:

$$[e_1{:}\rho_1, \ldots, e_n{:}\rho_n] \bullet e{:}\rho = [e_1{:}\rho_1, \ldots, e_n{:}\rho_n, e{:}\rho]$$

Metavariables, domains and evaluation judgements.

$e{:}\rho, h \Downarrow v$ Expression e in environment ρ evaluates to value v
$v, e{:}\rho, h \Downarrow v'$ Value v applied to expression e in environment ρ gives value v'
$\rho \in Env = Var \rightharpoonup Exp \times Env$
$v \in Value = \{e : \rho, h \mid h \in History, e \neq (\lambda x.e_0)e_1\}$
$h \in History = (Exp \times Env)^*$

Judgements now have the form $e : \rho, h \;\Downarrow\; e' : \rho', h'$ where h is the history *before* evaluating e, and h' is the history *after* evaluating e. Correspondingly, we redefine a value to be of form $v = e{:}\rho, h$ where e is not a β-redex.

$$(\mathrm{Lam}) \frac{}{\lambda x.e{:}\rho, h \Downarrow \lambda x.e{:}\rho,\; h{\bullet}(\lambda x.e{:}\rho)}$$

$$(\mathrm{Freevar}) \quad \frac{x \text{ free in } M}{x{:}\rho, h \;\Downarrow\; x{:}[],\; h{\bullet}(x{:}[])} \qquad (\mathrm{Boundvar}) \quad \frac{\rho(x),\; h{\bullet}(x{:}\rho) \;\Downarrow\; v}{x{:}\rho, h \;\Downarrow\; v}$$

$$(\mathrm{AP}) \quad \frac{e_1{:}\rho,\; h{\bullet}(e_1 @ e_2, \rho) \Downarrow\; v_1 \qquad v_1, e_2{:}\rho \;\Downarrow\; v}{e_1 @ e_2{:}\rho, h \;\Downarrow\; v}$$

$$(\mathrm{AP}_\lambda)\quad \frac{e' = \lambda x.e'' \qquad \rho'' = \rho'[x \mapsto e_2{:}\rho] \qquad e''{:}\rho'', h' \ \Downarrow\ v}{e'{:}\rho', h',\ e_2{:}\rho \ \Downarrow\ v}$$

$$(\mathrm{AP}_{\overline{\lambda}})\quad \frac{e' \neq \lambda x.e'' \qquad e_2{:}\rho, h' \Downarrow e_2'{:}\rho_2', h_2' \qquad \mathrm{fv}(e') \cap \mathrm{dom}(\rho_2') = \emptyset}{e'{:}\rho', h',\ e_2{:}\rho \Downarrow (e'@e_2'){:}\rho_2', h_2'}$$

Histories are accumulative It is easy to verify that h is a prefix of h' whenever $e{:}\rho, h \Downarrow e'{:}\rho', h'$.

4.3 Making environments nonrecursive

The presence of the history makes it possible to bind a variable x not to a pair $e{:}\rho$, but instead to the position of a prefix of history h. Thus $\rho \in Env = Var \to \mathbb{N}$. Domains and evaluation judgement:

$e{:}\rho, h \Downarrow v$ Expression e in environment ρ evaluates to value v
$v, e{:}\rho, h \Downarrow v'$ Value v applied to argument e in environment ρ gives value v'
$\rho \in Env = Var \rightharpoonup \mathbb{N}$
$h \in History = (Exp \times Env)^*$

A major difference: environments are now "flat" (nonrecursive) since Env is no longer defined recursively. Nonetheless, environment access is still possible, since at all times the current history includes all previously traversed expressions.

Only small changes are needed, to rules (AP_λ) and (Boundvar); the remaining are identical to the previous version and so not repeated.

$$(\mathrm{AP}_\lambda)\quad \frac{e' = \lambda x.e'' \qquad \rho'' = \rho'[x \mapsto |h|] \qquad e''{:}\rho'', h' \ \Downarrow\ v}{e'{:}\rho', h',\ e_1@e_2{:}\rho, h \ \Downarrow\ v}$$

$$(\mathrm{Boundvar})\quad \frac{nth\ \rho(x)\ h = e_1@e_2{:}\rho' \qquad e_2{:}\rho' \Downarrow v}{x{:}\rho, h \ \Downarrow\ v}$$

In rule (AP_λ), variable x is bound to length $|h|$ of the history h that was current for $(\mathtt{e}_1@\mathtt{e}_2, \rho)$. As a consequence bound variable access had to be changed to match. The indexing function $nth : \mathbb{N} \to History \to Exp \times Env$ is defined by: $nth\ i\ [e_1{:}\rho_1, \ldots, e_n{:}\rho_n] = e_i{:}\rho_i$.

4.4 Weak UNP: back pointers and no environments

This version is a semantics completely free of environments: it manipulates only traversals and back pointers to them. How it works: it replaces an environment by two back pointers, and finds the value of a variable by looking it up in the history, following the back pointers.

Details will be appear in a later version of this paper

5 The low-level residual language LLL

The semantics of Section 4.4 manipulates first-order values. We abstract these into a tiny first-order functional language called LLL: essentially a machine language with a heap and recursion, equivalent in power and expressiveness to the language F in book [11].

Program variables have simple types (not in any way depending on M). A token, or a product type, has a static structure, fixed for any one LLL program. A list type `[tau]` denotes dynamically constructed values, with constructors `[] and :`. Deconstruction is done by `case`. Types are as follows, where Token denotes an atomic symbol (from a fixed alphabet).

```
tau ::=  Token  |  (tau, tau)  |  [ tau ]
```

Syntax of LLL

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

6 Interpreters, compilers, compiler generation

Partial evaluation (see [12]) can be used to specialise a normalisation algorithm to the expression being normalised. The net effect is to compile an ULC expression into an LLL equivalent that contains no ULC-syntax; the target programs are first-order recursive functional program with "cons". Functions have only a fixed number of arguments, independent of the input λ-expression M.

6.1 Partial evaluation (= program specialisation)

One goal of this research is to partially evaluate a normaliser with respect to "static" input M. An effect can be to compile ULC into a lower-level language.

Partial evaluation, briefly A partial evaluator is a *program specialiser*, called *spec*. Its defining property:

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

The net effect is a *staging transformation*: $[\![p]\!](s, d)$ is a 1-stage computation; but $[\![[\![\mathit{spec}]\!](p, s)]\!](d)$ is a 2-stage computation.

Program speedup is obtained by *precomputation*. Given a program p and "static" input value 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, d.

The concept is historically well-known in recursive function theory, as the S-1-1 theorem. In recent years partial evaluation has emerged as the practice of engineering the S-1-1 theorem on real programs [12]. One application is compiling. Further, self-application of *spec* can achieve compiler generation (from an interpreter), and even compiler generator generation (details in [12]).

6.2 Should normalisation be staged?

In the current λ-calculus tradition M is self-contained; there is no dynamic data. So why would one wish to break normalisation into 2 stages?

Some motivations for staging The specialisation definition looks almost trivial on a normaliser program NP:

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

An extension: allow M to have separate input data, e.g., the input value $\underline{2}$ as in the example of Section 2. Assume that NP is extended to allow run-time input data.[7] The specialisation definition becomes:

$$\forall M \in \Lambda, d \in Data \,.\, [\![\, [\![spec]\!](\mathrm{NP}, M)]\!](d) = [\![\mathrm{NP}]\!](M, d) =_\beta M@d$$

Is staged normalisation a good idea? Let $\mathrm{NP}_M = [\![spec]\!](\mathrm{NP}, M)$ be the specialiser output.

1. One motivation is that NP_M can be in a much simpler language than the λ-calculus. Our candidate: the "low-level language" LLL of Section 5.
2. A well-known fact: the traversal of M may be much larger than M. By Statman's results [19] it may be larger by a "non-elementary" amount (!). Nonetheless it is possible to construct a λ-free residual program NP_M with $|NP_M| = O(|M|)$, i.e., such that M's LLL equivalent has *size that is only linearly larger* than M itself. More on this in Section 6.3.
3. A next step: consider *computational complexity* of normalising M, if it is applied to an external input d. For example the Church numeral multiplication algorithm runs in time of the order of the product of the sizes of its two inputs.
4. Further, two stages are natural for semantics-directed compiler generation.

[7] Semantics: simply apply M to Church numeral d before normalisation begins.

How to do staging Ideally the partial evaluator can do, at specialisation time, all of the NP operations that depend only on M. As a consequence, NP_M will have *no operations at all* to decompose or build lambda expressions while it runs on data d. The "residual code" in NP_M will contain only operations to extend the current traversal, and operations to test token values and to follow the back pointers.

Subexpressions of M may appear in the low-level code, but are only used as indivisible tokens. They are only used for equality comparisons with other tokens, and so could be replaced by numeric codes – tags to be set and tested.

6.3 How to specialise NP with respect to M ?

The first step is to *annotate* parts of (the program for) NP as either static or dynamic. Computations in NP will be either *unfolded* (i.e., done at partial evaluation time) or *residualised*: Runtime code is generated to do computation in the output program $\text{NP}_{\texttt{mul}}$ (this is p_s as seen in the definition of a specialiser).

Static: Variables ranging over *syntactic objects* are annotated as static. Examples include the λ-expressions that are subexpressions of M. Since there are only finitely many of these for any fixed input M, it is safe to classify such syntactic data as static.

Dynamic: Back pointers are dynamic; so the traversal being built must be dynamic too. One must classify data relevant to traversals or histories as dynamic, since there are unboundedly many.[8]

For specialisation, all function calls of the traversal algorithm to itself that do not progress from one M subexpression to a proper subexpression are annotated as "dynamic". The motivation is increased efficiency: no such recursive calls in the traversal-builder will be unfolded while producing the generator; but *all other calls* will be unfolded.

About the size of the compiled λ-expression (as discussed in Section 6.2).

We assume that NP is *semi-compositional*: static arguments e_i in a function call $f(e_1, \ldots, e_n)$ must either be absolutely bounded, or be substructures of M (and thus λ-expressions).

The size of NP_M will be linear in $|M|$ if for any NP function $f(x_1, \ldots, x_n)$, each static argument is either completely bounded or of BSV; and there is at most one BSV argument, and it is always a subexpression of M.

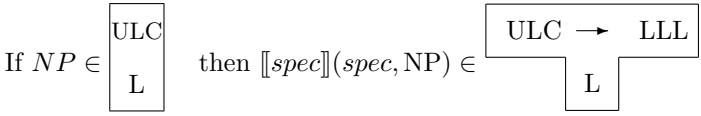
[8] In some cases more can be made static: "the trick" can be used to make static copies of dynamic values that are of BSV, i.e., of *bounded static variation*, see discussion in [12].

6.4 Transforming a normaliser into a compiler

Partial evaluation can transform the ULC (or STNP) normalisation algorithm NP into a program to compute a semantics-preserving function

$$f : \text{ULC} \to \text{LLL (or f : STLC} \to \text{LLL)}$$

This follows from the second Futamura projection. In diagram notation of [12]:



Here L is the language in which the partial evaluator and normaliser are written, and LLL of Section 5 is a sublanguage large enough to contain all of the dynamic operations performed by NP.

Extending this line of thought, one can anticipate its use for a *semantics-directed compiler generator*, an aim expressed in [10]. The idea would be to use LLL as a general-purpose intermediate language to express semantics.

6.5 Loops from out of nowhere

Consider again the Church numeral multiplication (as in Figure 1), but with a difference: suppose the data input values for m, n are given separately, at the time when program $\text{NP}_{\texttt{mul}}$ is run.. Expectations:

- Neither `mul` nor the data contain any loops or recursion. However `mul` will be compiled into an *LLL -program* $\text{NP}_{\texttt{mul}}$ *with two nested loops.*
- Applied to two Church numerals m, n, NP $_{\texttt{mul}}$ computes their product by doing one pass over the Church numeral for m, interleaved with m passes over the Church numeral for n. (One might expect this intuitively).
- These appear as an artifact of the specialisation process. The reason the loops appear: While constructing $\text{NP}_{\texttt{mul}}$ (i.e., during specialisation of NP to its static input `mul`), the specialiser will encounter the same static values (subexpressions of M) more than once.

7 Current status of the research

Work on the simply-typed λ-calculus

We implemented a version of STNP in HASKELL and another in SCHEME. We plan to use the UNMIX partial evaluator (Sergei Romanenko) to do automatic partial evaluation and compiler generation. The HASKELL version is more complete, including: typing; conversion to eta-long form; the traversal algorithm itself; and construction of the normalised term.

We have handwritten STNP-gen in SCHEME. This is the *generating extension* of STNP. Effect: compile from UNC into LLL, so $NP_M = [\![\text{STNP-gen}]\!](M)$. Program STNP-gen is essentially the compiler generated from STNP that could be obtained as in Section 6.4 Currently STNP-gen yields output NP_M as a `scheme` program, one that would be easy to convert into LLL as in Section 5.

Work on the untyped λ-calculus

UNP is a normaliser for UNC. A single traversal item may have two back pointers (in comparison: STNP uses one). UNP is defined semi-compositionally by recursion on syntax of ULC-expression M. UNP has been written in HASKELL and works on a variety of examples. A more abstract definition of UNP is on the way, extending Section 4.4.

By specialising UNP, an arbitrary untyped ULC-expression can be translated to LLL . A correctness proof of UNP is pending. No SCHEME version or generating extension has yet been done, though this looks worthwhile for experiments using UNMIX.

Next steps

More needs to be done towards separating programs from data in ULC (Section 6.5 was just a sketch). A current line is to express such program-data games in a *communicating* version of LLL . Traditional methods for compiling *remote function calls* are probably relevant.

It seems worthwhile to investigate *computational complexity* (e.g., of the λ-calculus); and as well, the *data-flow analysis* of output programs (e.g., for program optimisation in time and space).

Another direction is to study the utility of LLL as an intermediate language for a semantics-directed compiler generator.

References

1. S. Abramsky and G. McCusker. Game semantics. In *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, pages 1–56. Springer Verlag, 1999.
2. Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, pages 1–15, 1994.
3. Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.
4. Daniil Berezun. UNP: normalisation of the untyped λ-expression by linear head reduction. *ongoing work*, 2016.
5. William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logic Methods in Computer Science*, 5(1), 2009.
6. William Blum and Luke Ong. A concrete presentation of game semantics. In *Galop 2008:Games for Logic and Programming Languages*, 2008.

7. Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of girard's execution formula). In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pages 296–306, 1993.
8. Vincent Danos and Laurent Regnier. Head linear reduction. *unpublished*, 2004.
9. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.
10. Neil D. Jones, editor. *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980*, volume 94 of *Lecture Notes in Computer Science*. Springer, 1980.
11. Neil D. Jones. *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press, 1997.
12. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.
13. Andrew D. Ker, Hanno Nickau, and C.-H. Luke Ong. Innocent game models of untyped lambda-calculus. *Theor. Comput. Sci.*, 272(1-2):247–292, 2002.
14. Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 353–364, 2012.
15. C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 81–90, 2006.
16. C.-H. Luke Ong. Normalisation by traversals. *CoRR*, abs/1511.02629, 2015.
17. Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.
18. David A. Schmidt. State transition machines for lambda calculus expressions. In Jones [10], pages 415–440.
19. Richard Statman. The typed lambda-calculus is not elementary recursive. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, pages 90–94, 1977.