Compiling Untyped Lambda Calculus to Lower-Level Code by Game Semantics and Partial Evaluation (Invited Paper)

Daniil Berezun
JetBrains and St. Petersburg State University, Russian Federation
daniil.berezun@jetbrains.com

Neil D. Jones
DIKU, University of Copenhagen, Denmark
neil@diku.dk

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 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.

Categories and Subject Descriptors F. Theory of Computation [*F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: F.4.1 Mathematical Logic

General Terms THEORY, LANGUAGES, ALGORITHMS

Keywords Normalisation, traversal, game, untyped lambda calculus, semantics, partial evaluation, program transformation

1. Context and contribution

Plotkin posed the problem of existence of a fully abstract semantics of PCF [1]. Game semantics provided the first solution [2–5]. 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" or "thunks" for function calls and parameters. It looks promising to apply this new game semantics viewpoint to see its operational consequences. Further, it may give a new angle of attack on an old topic: *semantics-directed compiler generation* [6, 7].

Basis: λ-expression syntax is $e ::= x \mid e_1 @ e_2 \mid \lambda x.e$. Free and bound variables are defined as usual. ULC and STLC, respectively, denote the untyped and the simply typed λ-calculus.

Paper [8] uses the game semantics framework to develop an STLC normalisation procedure, using the traversal concept from [9, 10]. We call this procedure STNP for short.

The STNP algorithm in [8] 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 $\kappa \vdash M : A$ where A is a type and κ is a type environment.

Paper [8] contains a complete correctness proof. The 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.

STNP can be seen as an *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: We develop a traversal-based algorithm for ULC in a systematic, semantics-directed way. Our approach differs from and is simpler than [11]. We explain briefly how partial evaluation can be used to implement ULC, compiling it to a low-level language.

2. Normalisation by traversal

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 [8–10] 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 maybe visited more than once, some not at all.)

An example: multiplication of Church numerals[1]

The Church-Turing thesis: a function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is partial recursive (computable) iff there is a λ-expression M with free variables S, Z such that for any $x_1, \ldots, x_n, x \in \mathbb{N}$

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

Multiplication is an example:

$$\texttt{mul} = \lambda m.\lambda n.m(nS)Z$$

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

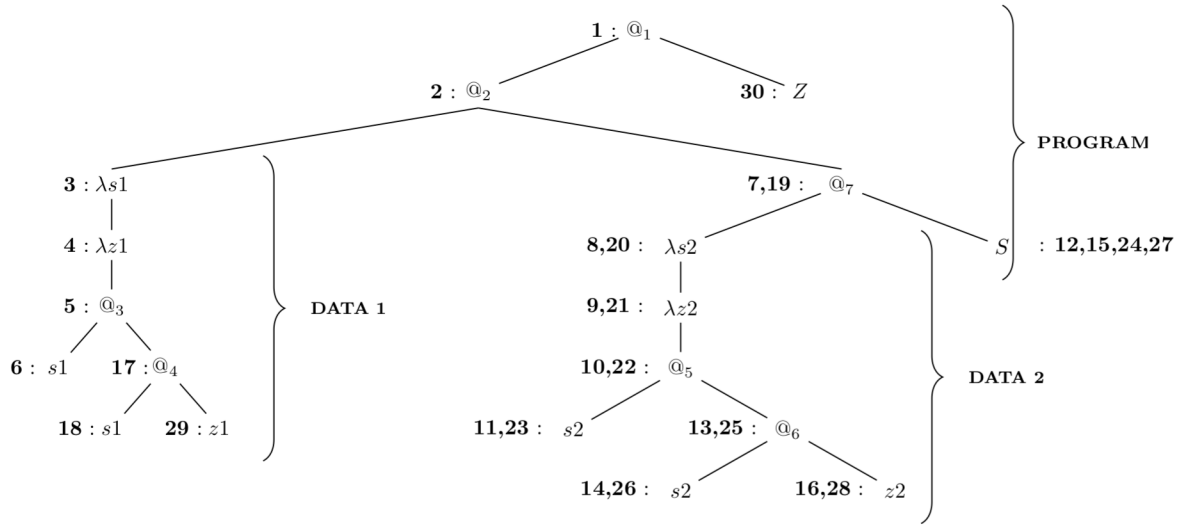


Figure 1: Traversal of $\underline{2}(\underline{2}S)Z$.

An instance: application `mul` $\underline{2}$ $\underline{2}$ reduces to S^4Z.

The unique traversal of $\underline{2}\,(\underline{2}\,S)\,Z$ visits each subexpression of Church numeral $\underline{2}$ once. However it visits the subexpressions of $\underline{2}$ twice, since in general $x * y$ is computed by adding y together x times. The normal form of $\underline{2}\,(\underline{2}\,S)\,Z$ is S^4Z: the core of Church numeral $\underline{4}$.

Figure 1 shows traversal of expression $\underline{2}\,(\underline{2}\,S)\,Z$ in tree form[2]. The labels **1:, 2:** etc. indicate the order in which subexpressions are traversed.

Computation by traversal can be seen as a game

The traversal in Figure 1 can be seen as a play between program $\lambda m\lambda n.(m@\ (n@S))@Z$ and the two data values $\underline{2}$ and $\underline{2}$).

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.

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. Evaluation by traversal

We now develop a type-free normalisation procedure UNP for ULC. In brief: UNP is to ULC as STNP is to STLC.

Differences between UNP and STNP as in [8]. 1: the simply typed λ-calculus is strongly normalising. However an untyped λ-expression may not have a normal form, in which case UNP evaluation does not terminate. 2: the STNP algorithm begins by converting its input λ-expression to η-long form. This is not relevant to UNP since its input is untyped. 3: STNP chooses the right traversal by using at most one *back pointer* for each traversed program node. The extra power of UNP (Turing completeness) comes with a price: *two kinds of back pointer* are used, one to manage control, and one to manage name binding.

For clarity we develop UNP by a series of evaluators. Each realises a semantics for the λ-calculus. The series starts with a traditional rewriting semantics, and ends with a traversal semantics generating the sequence seen in Figure 1. All are supported by Haskell programs (very similar to the evaluators here, and available online).

A canonical traversal order

How did Figure 1 select the right sequence of node visits to evaluate $\underline{2}(\underline{2}S)Z$? There are a great many possible traversals, mostly *not* corresponding to evaluation of M.

Danos and Regnier s *head linear reduction* yields normal forms when they exist [12], so we choose this as the *canonical traversal order* on input λ-expression M. For brevity we henceforth call a subexpression of M a *token*.

The canonical traversal order visit tokens in the order in which they are seen in complete linear head β-reduction. The example in Figure 1 has canonical traversal order:

$$@_1, @_2, \lambda s1, \lambda z1, @_3, s1, @_7, \lambda s2, \ldots, S, z2, z1, Z$$

A series of evaluators

We develop several evaluation semantics that all follow the canonical traversal order. Semantics 1 is classical β-reduction (a deterministic version). This is progressively transformed into more operationally interesting versions, ending in Semantics 5.

Semantics 5 is analogous to [8] for STLC, but now for all of ULC. It has surprisingly little algorithmic machinery, consisting of two kinds of back pointers, a "cons" operation, and a finite set of tokens. Only the token set is dependent on input λ-expression M.

Semantics 5 is the essence of an implementation of a prototypical functional programming language, and in it one can discern counterparts of some usual implementation machinery, in particu-

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

lar static links and dynamic links, and the control stack. Evaluation by Semantics 5 does not use environments, thunks, or closures.

Semantics-based stepping stones

The first evaluator does (a slightly optimised version of) the usual leftmost outermost β-reduction. The second is an environment-based semi-compositional version of the same; and the third, the result of converting this to continuation-passing style and defunctionalising. The result is tail-recursive. These transformations are standard.

The next step is to enrich the tail-recursive evaluator by adding a "history" argument to the evaluation function. This records the "traversal until now", and allows removal of the records added by defunctionalising; each is replaced by a control pointer. The final step obtains Semantics 5 by removing the recursively-defined environments; each is replaced by a binder pointer. The resulting evaluator generates traversals as in Figure 1 with surprisingly little computational machinery.

3.1 Substitution-based semantics

By definition e is in normal form iff it contains no redexes: subexpressions of form $e_1@e_2$ with $e_1 = \lambda x.e$. A well-known fact : the normal form (if it exists) can be obtained by leftmost-outermost reductions [13]. One way to normalise an application $e_1@e_2$: First evaluate e_1 using a *weak reduction* that doesn't reduce terms under abstraction; this yields a *weak normal form* $\lambda x.e$ for e_1'a value. Then apply β, substituting e_2 for x in e. See more details in [13, 14].

The semantics of weak reduction is as follows[3] [14]:

$$\frac{}{x \overset{wnf}{\Downarrow} x} \qquad \frac{}{\lambda x.e \overset{wnf}{\Downarrow} \lambda x.e}$$

$$\frac{e_1 \overset{wnf}{\Downarrow} \lambda x.e \quad e[e_2/x] \overset{wnf}{\Downarrow} e'}{e_1@e_2 \overset{wnf}{\Downarrow} e'} \qquad \frac{e_1 \overset{wnf}{\Downarrow} e' \not\equiv \lambda x.e}{e_1@e_2 \overset{wnf}{\Downarrow} e'_1@e_2}$$

Using this basis, strong reduction to normal form can be done by the following semantic rules [14]:

$$\frac{}{x \overset{snf}{\Downarrow} x} \qquad \frac{e_1 \overset{wnf}{\Downarrow} \lambda x.e \quad e[e_2/x] \overset{snf}{\Downarrow} e'}{e_1@e_2 \overset{snf}{\Downarrow} e'}$$

$$\frac{e \overset{snf}{\Downarrow} e'}{\lambda x.e \overset{snf}{\Downarrow} \lambda x.e'} \qquad \frac{e_1 \overset{wnf}{\Downarrow} e1' \not\equiv \lambda x.e \quad e1' \overset{snf}{\Downarrow} e1'' \quad e_2 \overset{snf}{\Downarrow} e'_2}{e_1@e_2 \overset{snf}{\Downarrow} e''_1@e'_2}$$

A procedure to implement the rules above follows, using $wnf, snf : Exp \rightharpoonup Exp$: strongly

$$\begin{array}{l} wnf(x) = x, \qquad wnf(\lambda x.e) = \lambda x.e, \\ wnf(e_1@e_2) = wnf(e[e_2/x]) \text{ if } wnf(e_1) = \lambda x.e \\ wnf(e_1@e_2) = wnf(e_1) \mathbin{@} e_2 \text{ otherwise} \end{array}$$

$$\begin{array}{l} snf(x) = x, \qquad snf(\lambda x.e) = \lambda x.snf(e), \\ snf(e_1@e_2) = snf(e[e_2/x]) \text{ if } wnf(e_1) = \lambda x.e \\ snf(e_1@e_2) = snf(e'_1) \mathbin{@} snf(e_2) \text{ otherwise} \\ \qquad\qquad \text{where } wnf(e_1) = e'_1 \end{array}$$

This normalisation procedure snf is an effective reduction strategy (as that term is used in [13]). It is not tail recursive; and evaluation involves using variable names at run-time. Neither property is good for compilation.

[3] $e[e'/x]$ is the result of replacing all free occurrences of x in e by e', renaming as needed to avoid name capture.

3.2 Evaluation with environments

This evaluator resembles that of Section 3.1, but uses an environment ρ to maintain a record of substitutions. This and remaining steps use the style of denotational semantics:

Syntactic arguments of the semantic equations are enclosed in semantic brackets $[\![\_]\!]$, and ρ is an environment. We write $A \rightharpoonup B$ for the domain of *partial* functions from A to B. Usage: $\mathcal{R}[\![e]\!] = \bot$ if program e does not terminate.

Domains : (and initial environment)

$$\begin{array}{lcl} e \in Exp & = & \lambda - expression \\ EE & = & Exp \times Env \\ \rho \in Env & = & Variable \to EE \cup \{Free\} \\ \rho_0 & = & \lambda x.Free \\ \alpha \in Flag & = & \{T, F\} \text{ (application context)} \end{array}$$

Semantic functions :

$$\begin{array}{lcl} \mathcal{R} : & Exp & \rightharpoonup EE \\ [\![\ ]\!] : & Exp & \to Flag \to Env \rightharpoonup EE \end{array}$$

Semantic equations:

$$\begin{array}{lcl} \mathcal{R}[\![e]\!] & = & [\![e]\!]\ F\ \rho_0 \\ [\![x]\!]\ \alpha\ \rho & = & \textbf{case}\ \rho\ x\ \textbf{of}\ (e', \rho') \Rightarrow [\![e']\!]\ \alpha\ \rho' \\ & & \qquad\qquad Free \quad \Rightarrow (x, \rho_0) \\ [\![\lambda x.e]\!]\ T\ \rho & = & (\lambda x.e, \rho) \\ [\![\lambda x.e]\!]\ F\ \rho & = & [\![e]\!]\ F\ \rho[x \mapsto Free] \end{array}$$

$$\begin{array}{lcl} [\![e_1@e_2]\!]\ \alpha\ \rho = & \multicolumn{2}{l}{\textbf{let}\ (e'_1, \rho') = [\![e_1]\!]\ T\ \rho\ \textbf{in}} \\ & \multicolumn{2}{l}{\textbf{case}\ e'_1\ \textbf{of}} \\ & \multicolumn{2}{l}{\quad \lambda x.e_0 \Rightarrow [\![e_0]\!]\ \alpha\ \rho'[x \mapsto (e_2, \rho)]} \\ & \multicolumn{2}{l}{\quad v \quad\ \Rightarrow [\![e_2]\!]\ F\ \rho} \end{array}$$

The argument α (application context) is a flag, set to T for weak evaluation, F for strong evaluation.

This semantics is not compositional[4] for application @ and for bound variables: in both cases an argument of $[\![\ ]\!]$ is not a syntactic substructure of the semantic equation's left side.

On the other hand, this semantics has a *semi-compositional* property that is essential for compiling efficient target code. (Terminology from [15]: function $[\![\ ]\!]$ is only applied to arguments that are syntactic substructures of *the λ-expression currently being evaluated*. A consequence: semantic variable e is of "bounded static variation".)

Constructing an output λ-expression: Semantics 2 yields as output a pair $(e', \rho) = \mathcal{R}[\![e]\!]$. For conciseness no output λ-expression syntax is constructed, i.e., the purpose of Semantics 2 is only to see whether a normal form exists.

It is not difficult to extend Semantics 2 to actually compute the normal form, and we have implemented it in Haskell. The result, however, is more complex, as a result (e', ρ) has to be converted into output into a λ-expression by unfolding environments, and renaming variables where necessary.

3.3 Tail-recursive evaluation

The next step is to remove the nested function calls in the above semantics, yielding a *tail recursive* version that is closer to low-level code. To do this: first, add continuation functions in place of the nested calls. Second, replace the continuation functions by data structures, using the well-known Reynolds' *defunctionalisation* transformation [16].

Following is the result of defunctionalising a continuation semantics. $k \in K$ is a defunctionalised continuation (a data values).

[4] Compositionality (as used in denotational semantics): the denotation of every syntactic construction is a combination of the denotations of its syntactic substructures.

Function apk applies k to an expression value. The output (if any) is **Succeed**: the input has been successfully traversed. (Again, the normal form is not constructed.)

Domains :

$$
\begin{array}{lcll}
e \in Exp & = & \lambda - Expression & \\
EE & = & Exp \times Env & \\
\rho \in Env & = & Variable \rightharpoonup EE \cup \{Free\} & \\
\rho_0 & = & \lambda x.Free & \\
\alpha \in Flag & = & \{T, F\} & \text{(application context)} \\
k \in K & = & \{\langle Kend \rangle\} & \text{(continuations)} \\
 & \cup & \{\langle Kapp\, e\, \alpha\, \rho\, k \rangle \mid e \in Exp, \rho \in Env, k \in K\} &
\end{array}
$$

Semantic functions :

$$
\begin{array}{lllll}
\mathcal{R}: & Exp & \rightharpoonup & \{\mathbf{Succeed}\} & \\
[\![\,]\!]: & Exp & \rightarrow & Flag \rightarrow Env \rightarrow K \rightharpoonup \{\mathbf{Succeed}\} & \\
apk: & Exp & \rightarrow & K \rightarrow Env \rightharpoonup \{\mathbf{Succeed}\} &
\end{array}
$$

Semantic equations:

$$
\begin{array}{lcl}
\mathcal{R}[\![e]\!] & = & [\![e]\!]\ F\ \rho_0\ \langle Kend \rangle \\
 & & \\
[\![x]\!]\ \alpha\ \rho\ k & = & \mathbf{case}\ \rho\, x\ \mathbf{of}\ \begin{array}{ll} Free & \Rightarrow apk\ x\ k\ \rho_0 \\ (e', \rho') & \Rightarrow [\![e']\!]\ \alpha\ \rho'\ k \end{array} \\
 & & \\
[\![e_1 @ e_2]\!]\ \alpha\ \rho\ k & = & [\![e_1]\!]\ T\ \rho\ \langle Kapp\ e_2\ \alpha\ \rho\ k \rangle \\
 & & \\
[\![\lambda x.e]\!]\ F\ \rho\ k & = & [\![e]\!]\ F\ \rho[x \mapsto Free]\ k \\
[\![\lambda x.e]\!]\ T\ \rho\ \langle Kapp\ e'\ \alpha\ \rho'\ k \rangle & = & [\![e]\!]\ \alpha\ \rho[x \mapsto (e', \rho')]\ k \\
[\![\lambda x.e]\!]\ T\ \rho\ \langle Kend \rangle & = & \mathbf{Succeed}
\end{array}
$$

$$
\begin{array}{lcl}
apk\ e\ \ \langle Kend \rangle\ \ \ \rho & = & \mathbf{Succeed} \\
apk\ e\ \langle Kapp\ e'\ \alpha\ \rho'\ k \rangle\ \rho & = & \mathbf{case}\ e\ \mathbf{of} \\
\quad \lambda x.e'' & \Rightarrow & [\![e'']\!]\ \alpha\ \rho[x \mapsto (e', \rho')]\ k \\
\quad \_ & \Rightarrow & [\![e']\!]\ F\ \rho'\ k
\end{array}
$$

Comment: The rule from Section 3.2 for $[\![e_1 @ e_2]\!]$ has been split into two: one part calls $[\![e_1]\!]$; and the other part appears in the definition of apk. Note that these may have quite different times of execution. This property is quite visible using continuations, but is not obvious in Section 3.2.

A consequence of tail recursion and semi-compositionality: Given an input λ-expression M, the rules above will perform a sequence of calls

$$[\![e_1]\!]\alpha_1\ \rho_1\ k_1 \rightarrow [\![e_2]\!]\alpha_2\ \rho_2\ k_2 \rightarrow \ldots \rightarrow [\![e_i]\!]\alpha_i\ \rho_i\ k_i \rightarrow \ldots$$

where $e_1 = M$, each e_i is a subexpression of M. Further, $e_1, e_2, \ldots$ is the canonical traversal order for M. The next evaluators further develop a linear evaluation style, so it more closely resembles target program execution.

3.4 Traversal with history and environment

The next-last step: replace the continuation values k by back-pointers to a *history* h. Corresponding to a Semantics 3 computation, Semantics 4 performs calls

$$[\![e_1]\!]h_1 \rightarrow [\![e_2]\!]h_2 \rightarrow \ldots \rightarrow [\![e_i]\!]h_i \rightarrow \ldots$$

A history h is an accumulative trace that remembers *which functions* of Semantics 3 were called with *which arguments*.

$$h \in H = (Exp \times Flag \times Env \times H)^*$$

Each history h_i is a list:[5] a record of *all calls* made to $[\![e_j]\!]$ for $j = 1, \ldots, i$. For $i > 0$ history item h_i has the form $h_i = \langle e_i\ \alpha_i\ \rho_i\ ch_i \rangle : h_{i-1}$, where ch_i represents continuation k_i. [6]

The idea is to replace a Semantics 3 continuation data structure such as $\langle (Kapp\, e_2)\, \alpha\, \rho\, k \rangle$ by the *the time t_i at which the continuation was created*. Operationally, time t_i is the back pointer component ch_i of the top item in history h_i.

Changes from Semantics 3: first, the history is an *accumulative* record. Second, each K value k_i has been replaced by a **control history** ch_i. This is a prefix of the current history, i.e., a back pointer to an earlier position in the history. Third, Semantics 4 need not allocate any continuation records at all, since the information in them is contained in the history items $\langle e\ \alpha\ \rho\ ch \rangle$.

Domains :

$$
\begin{array}{lcll}
e \in Exp & = & \lambda - Expression & \\
EE & = & Exp \times Env & \\
\rho \in Env & = & Variable \rightharpoonup EE \cup \{Free\} & \\
\rho_0 & = & \lambda x\ .\ Free & \\
\alpha \in Flag & = & \{T, F\} & \\
h, \in H, ch \in CH & = & [\ Item\ ] & \text{(History)} \\
it \in Item & = & \langle Exp\ Flag\ Env\ CH \rangle &
\end{array}
$$

Semantic functions :

$$
\begin{array}{l}
\mathcal{R} : Exp \rightharpoonup H \\
eval : H \rightharpoonup H \\
apk : Exp \rightarrow Env \rightarrow CH \rightarrow H \rightharpoonup H
\end{array}
$$

Semantic equations:

$$\mathcal{R}[\![e]\!] \ = \ eval\ [\ \langle\ e\ F\ \rho_0\ [\,]\ \rangle\ ]$$

$$
\begin{array}{l}
eval\ h = \mathbf{let}\ it : \_ = h\ \mathbf{in\ case}\ it\ \mathbf{of} \\
\quad \begin{array}{ll}
\langle \quad x \quad \alpha\ \rho\ ch \rangle \Rightarrow apk\ x\ \rho_0\ ch\ h & \mathbf{if}\ \rho\, x = Free \\
\langle \quad x \quad \alpha\ \rho\ ch \rangle \Rightarrow eval\ \langle e'\ \alpha\ \rho'\ ch \rangle : h & \mathbf{if}\ \rho\, x = (e', \rho') \\
\langle\ \lambda x.e \quad T\ \rho\ ch \rangle \Rightarrow apk\ \lambda x.e\ \rho\ ch\ h & \\
\langle\ \lambda x.e \quad F\ \rho\ ch \rangle \Rightarrow eval\ \langle e\ F\ \rho[x \mapsto Free]\ ch \rangle : h & \\
\langle e_1 @ e_2\ \alpha\ \rho\ ch \rangle \Rightarrow eval\ \langle e_1\ T\ \rho\ h \rangle : h &
\end{array}
\end{array}
$$

$$
\begin{array}{lcl}
apk\ e\ \rho\ ch\ h & = & \mathbf{case}\ ch\ \mathbf{of} \\
 & & \quad [\,] \qquad\qquad\qquad\quad \Rightarrow h \\
 & & \quad (\langle e_1 @ e_2\ \alpha\ \rho_0\ ch' \rangle : \_) \quad \Rightarrow\ eval\ (f\ e) : h \\
 & & \quad\ \mathbf{where} \\
 & & \qquad f\ (\lambda x.e_0) = \langle e_0\ \alpha\ \rho[x \mapsto (e_2, \rho)]\ \ ch' \rangle \\
 & & \qquad f \quad e \qquad\ = \langle e_2\ F\ \rho_0\ ch' \rangle
\end{array}
$$

3.5 UNP: Traversal with only a history

In this step each environment is replaced by a **binder history**. This is a prefix of the current history, and is named *bh*. A new function called $lookup$ realises the effect of an environment ρ by tracing links through *bh* to find the value of an argument x.[7]

In addition, in the subsection we assume that the input lambda expressions contain *deBruijn indexes*. Usually deBruijn indexes are used instead of variable names. In our case, we use both deBruijn indexes and variable names for the sake of readability: the residual traversal is much easier to be understand if variable names are used. Thus, we use deBruijn indexes to define semantics and variable names to appear in the residual traversal.

[5] We use a Haskell functional programming notation for sequences, e.g. $[v_1, \ldots, v_n]$, and $v : vs$ for "cons".

[6] Note that the expression e_i in a history item may come either from the program or from its data. Examples: nodes @1, @2 and $\lambda s1$, $\lambda s2$ in Figure 1.

[7] An analogy: the links for the binder history *bh* resemble the *static links* used in compiler implementations of block-structured or functional programs. The control back pointer *ch* corresponds to the *dynamic link*, similar to the *return address*. More on this in Section 6.

Below, $BV\ x\ i$ denotes bound variable x with deBruijn index i; and $FV\ x$ denotes free variable x.

Domains:

$$
\begin{array}{lll}
e \in Exp & = \lambda - Expression & \\
\alpha \in Flag & = \{T,\ F\} & \\
h \in H, ch \in CH, bh \in BH & = [Item] & (History) \\
it \in Item & = \langle Exp\ Flag\ BH\ CH \rangle &
\end{array}
$$

Semantic functions:

$$
\begin{array}{lcl}
\mathcal{R} & : & Exp \rightharpoonup H \\
eval & : & H \rightharpoonup H \\
evoperand & : & Item \to H \rightharpoonup H \\
apk & : & Exp \to CH \to H \rightharpoonup H \\
lookup & : & Int \to Flag \to BH \to CH \to H \rightharpoonup H
\end{array}
$$

Semantic equations:

$$\mathcal{R}[\![e]\!] \ = \ eval\ [\ \langle e\ F\ [\,]\ [\,]\rangle\ ]$$

$$
\begin{array}{l}
eval\ \ h = \textbf{let}\ it : \_ = h\ \textbf{in case}\ it\ \textbf{of} \\
\quad \langle\ (FV\ x)\ \alpha\ bh\ \ ch\ \rangle \Rightarrow apk\ (FV\ x)\ ch\ h \\
\quad \langle\ (BV\ x\ i)\ \alpha\ bh\ \ ch\ \rangle \Rightarrow lookup\ i\ \alpha\ bh\ ch\ h \\
\quad \langle\ \ \lambda x.e\ \ \ T\ bh\ \ ch\ \rangle \Rightarrow apk\ \lambda x.e\ ch\ h \\
\quad \langle\ \ \lambda x.e\ \ \ F\ bh\ \ ch\ \rangle \Rightarrow eval\ \langle e\ \ F\ bh\ ch\rangle : h \\
\quad \langle\ \ e_1 @ e_2\ \ \alpha\ bh\ \ ch\ \rangle \Rightarrow eval\ \langle e_1\ T\ bh\ h\ \rangle : h
\end{array}
$$

$$
\begin{array}{l}
lookup\ 0\ \alpha\ (\langle \_\ T\ \_\ ch' \rangle : \_)\ ch\ h = \textbf{case}\ ch'\ \textbf{of} \\
\quad \langle e\ \_\ bh\ \_\rangle : \_ \ \Rightarrow\ evoperand\ \langle e\ \alpha\ bh\ ch\rangle\ h \\
\quad \_ \quad \Rightarrow \textbf{case}\ ch\ \textbf{of} \\
\quad\quad [\,] \qquad\qquad\qquad \Rightarrow\ h \\
\quad\quad \langle ap\ \_\ bh''\ ch''\rangle : \_ \Rightarrow\ evoperand\ \langle ap\ F\ bh''\ ch''\rangle\ h \\
lookup\ 0\ \_\ (\langle \_\ F\ \_\ ch' \rangle : h')\ ch\ h = apk\ (BV\ \_\ 0)\ ch\ h \\
lookup\ i\ \alpha\ (\langle \_\ \_\ bh'\ \_\rangle : \_)\ \ ch\ h = lookup\ (i-1)\ \alpha\ bh'\ ch\ h
\end{array}
$$

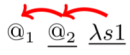
$$
\begin{array}{l}
apk\ \ \_\ \ [\,]\ h = h \\
apk\ \lambda x.e\ (\langle \_\ \alpha\ \ \_\ \ ch\rangle : \_)\ h = eval\ \langle e\ \alpha\ h\ ch\rangle : h \\
apk\ \ \_\ \ (\langle e\ \alpha\ bh\ ch\rangle : \_)\ h = evoperand\ \langle e\ F\ bh\ ch\rangle\ h
\end{array}
$$

$$evoperand\ \langle e_1 @ e_2\ \alpha\ bh\ ch\rangle\ h = eval\ \langle e_2\ \alpha\ bh\ ch\rangle : h$$

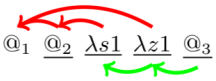
4. UNP: an example

We now revisit the multiplication example $mul\ \underline{2}\ \underline{2}$ from section 2. We will construct the output traversal as a justified sequence $h = h_1 h_2 \ldots h_{30}$. Each history node is equipped with two back pointers: *ch* for control; and *bh* for binding. In the following diagrams *bh* is the green lower back pointer and *ch* is the red upper back pointer. A token is underlined just in case its flag α has value T. The initial configuration is a root of the input term with empty back pointers, i.e. in our example initial $h = @_1$. We apply function *eval* to this argument.

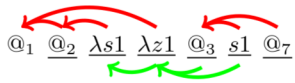
The first two steps are similar: the application-case of function *eval* is applied, each time we chose the leftmost branch, *bh* is the same as in the previous step (i.e. empty), and *ch* points to the current history.



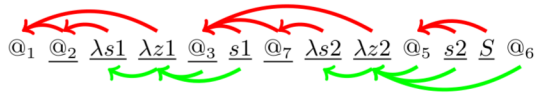
The next two steps are also similar: two lambda nodes are bound to the corresponding application nodes, i.e., continuation application function *apk* is called for both. Function *apk* adds the body of the lambda node to the current sequence and equips it with a binder pointer to the most recently bound lambda node. One element is "popped" from *bh*, and the *eval* function is called with unchanged flag α.



The next steps are similar to the first two; the one after that is more interesting. In this step a variable is seen (for the first time). This variable $s1$ is bound, and one can reach its binder via a sequence of two *bh* pointers (this is exactly what function *lookup* does). When the binder $\lambda s1$ is found, the *lookup* function returns a right child $@_7$ of the element binder pointed via the red pointer (*ch*), together with corresponding environment (i.e., the green *bh* pointer (empty in this case). Then *eval* is called again, on $@_7$.



The next interesting case is step number 12, when *eval* function calls $lookup\ h_{1..12}\ S$. Here S is free, so *apk* (apply continuation) is called. Since *ch* is not empty it calls *eval* on the right child of the last application whose right child has not yet been traversed, and which is not bound by some lambda node. This child is $@_6$.



Finally, the traversal will end at step 30. The whole traversal is presented in Figure 2.

5. Syntax of the low-level residual language LLL

A consequence of semi-compositionality: it is possible to compile any λ-expression M into a tiny first-order functional language that we call LLL. LLL is essentially a machine language with a heap and recursion, equivalent in power and expressiveness to the language F in book [17]. Section 6.4 shows how to compile M into LLL using partial evaluation.

For now, we briefly describe the target language LLL. The following is a tree grammar that generates a subset of LLL, one large enough to contain the target programs that are the compilations of lambda expressions. Nonterminal `h` denotes a history; and `item` denotes a history item. A program is a set of first-order function declarations.

A program variable has a simple type (not in any way depending on M). One type is the finite set of *tokens*, one for each subexpression of M. The Booleans are another finite type. There are two infinite types: lists (e.g., histories); and items (e.g., elements of histories).

Constructors `[]` (0-ary), `:` (binary) and `<. . .>` (4-ary) may form dynamically constructed values. Deconstruction is done by `case`. LLL also supports pattern matching of function arguments. The formal definition of LLL syntax can be found on Figure 3.

6. Interpreters, compilers, compiler generation

Partial evaluation can be used to specialise a normalisation algorithm to the expression being normalised (see [18]). Applied to UNP, partial evaluation compiles an ULC expression M into a low-level equivalent that contains no λ-syntax: The target program UNP_M for a source λ-expression will be a first-order recursive functional program.

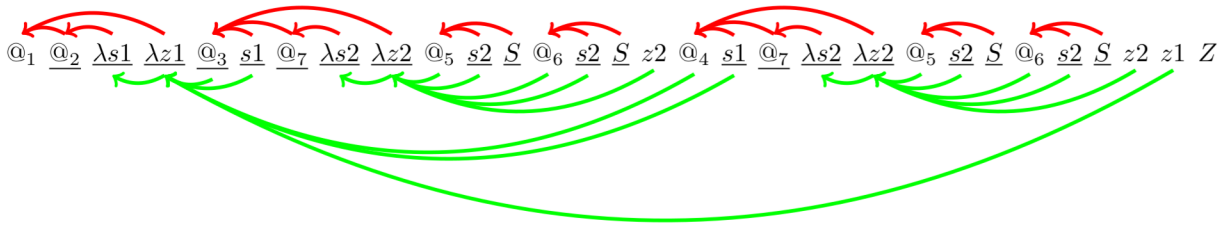


Figure 2: Traversal for $\underline{2}(\underline{2}S)Z$.

```
program ::=  def1  ...  defn

def   ::= f param1 ... paramn = e
param ::= x | token | b | [] | < tx bx x y > : z

tx    ::= x | token
bx    ::= x | b

e     ::= x | item | h | call
        | b | token | e : e | item        constructors
        | case e of param->e1...param->en  destructors
call  ::= f e1 ... en
item  ::= < token b h h >

x, y, z ::= variables
h       ::= []   | item : h
b       ::= True | False

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

Figure 3: Syntax of LLL language

6.1 Partial evaluation (program specialisation)

Partial evaluation, briefly

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

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

Net effect: a *staging transformation*. The run $[\![p]\!](s, d)$ is a 1-stage computation; but $[\![[\![spec]\!](p, s)]\!](d)$ describe two runs, i.e., a 2-stage computation.

Precomputation gives program speedup: $p_s \stackrel{def}{=} [\![spec]\!](p, s)$ is the *residual program* output produced by *spec*, when given a program p and its known "static" input data s. When run on any remaining "dynamic" data d, residual program p_s computes what p would have computed, if given *both* data inputs s and d.

The concept is historically well-known as the S-1-1 theorem in recursive function theory (though that theory did not study program speedup). In recent years partial evaluation has emerged as the practice of engineering the S-1-1 theorem on real programs [18]. One application area is compiling by specialising an interpreter. The idea is to regard its input program as static, and the input program's data as dynamic.

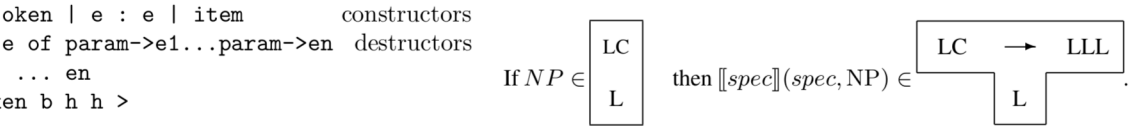
Further, by the *Futamura projections*, self-application of *spec* can achieve compiler generation (from an interpreter), and even compiler generator generation (details in [18]).

6.2 Transforming a normaliser into a compiler

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

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

This follows from the second Futamura projection. In the diagram notation of [18]:



Here LC is ULC or STLC; L is the language in which the partial evaluator and normaliser are written; and LLL (e.g., as in Section 5) is a sublanguage of L that is large enough to contain all the dynamic operations done by NP.

Extending this line of thought, one can anticipate its use for a *semantics-directed compiler generator*, an aim expressed in [6]. The idea would be to use LLL as a general-purpose intermediate language to express semantics.

6.3 About the result of specialising UNP to M

A partial evaluator will, while specialising UNP to M, perform of the UNP operations that depend only on M. As a consequence, UNP_M will have *no operations at all* to decompose or build lambda expressions while it runs on data d. The residual program UNP_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.

6.4 How to specialise UNP with respect to M ?

The first step is *binding time analysis*: to annotate all parts of (the program for) UNP in Section 3.5 as either static or dynamic. (This is done independently of M.) Computations in UNP are marked to be either *unfolded* (i.e., done at partial evaluation time) or *residualised*: runtime code is generated to do computation in the output program UNP_M (this is the p_s seen in the definition of a specialiser).

Static: some UNP variables of *bounded static variation* (a term from [18]) will be annotated as "static". The main example is e, ranging over subexpressions of M. This takes on only finitely many

vales for any fixed input M, so it is safe to classify it as static. Boolean flags are also static.[8]

Dynamic: Back pointers are not statically computable; so the traversal (i.e., history) being built is annotated as dynamic. Values obtained from histories are also annotated as dynamic: their sizes can be unbounded (for a given M).

For specialisation we annotate all recursive function calls within the traversal algorithm that do not progress from one M subexpression to a proper subexpression as "dynamic". The aim is both termination and efficiency: no dynamic recursive calls in the traversal-builder will be unfolded while specialising; *all other calls* will be unfolded.

7. Binding-time annotated UNP

We now show the result of a *binding-time annotation* of UNP. Assumption: we wish to specialise UNP to a static (fixed, known) lambda expression M. The following is the result of classifiying every expression in UNP as static or dynamic. A static value is constant or computable by the specialiser from (the syntax of) M. A dynamic value must be computed at run time, so residual code will be generated.

Further, each function call in M will be classified as either to be *unfolded*, i.e., performed at specialisation time; or *residual*. If residual, a specialised version of this call will appear in the output program resulting from specialisation. If a UNP subexpression is to be residualised, then an underline is added (see below for an example).

We use a box notation $\boxed{\ }$ to enclose static information. Examples:

- $\boxed{Exp}$ is a static type.
- For a bound variable, $\boxed{\underline{lookup\ i\ \alpha}}$ denotes a residual call. It could be specialised to static values $i = 2$ and $\alpha = T$, to yield a residual function $lookup_{2,T}\ bh\ ch\ h$.
- The call in the function *lookup*, $\boxed{i-1}$ is a static expression; and recursive call $\boxed{lookup\ i\ \alpha}$ inside *lookup* is a function call with two static arguments. (Static, and to be unfolded at specialisation time.)

7.1 Notations used for binding-time annotation

Annotation of function definitions and calls: a function type $f : V_1 \to \ldots \to V_i \to \ldots \to V_n$, where the first i arguments are static, will be annotated as

$$f : \boxed{V_1 \to \ldots V_i \to} V_{i+1} \to \ldots \to V_n$$

(In Semantics 5 static values occur only as prefixes of argument lists.)

On the right side of a function definition $f\ v_1 \ldots v_n = e$, the name and the static arguments of a call $g\ e_1 \ldots e_m$ will be enclosed in a box, i.e., $\boxed{g\ e_1 \ldots e_m}$. If this part is to be a residual call, then an underline is added: $\boxed{\underline{g\ e_1 \ldots\ e_i}}$. If the call in this box is *not underlined*, then the call will be unfolded, that is, performed by the partial evaluator at specialisation time.

In terms of *computational content*, the following is nearly identical to UNP (Semantics 5). There is one change, however, better to separate the binding times: Semantics 5 used function $eval : H \rightharpoonup H$ where $H = [Item]$.

We now retype $eval : H \rightharpoonup H$ by unrolling the H type, and separating the components of the beginning $Item$. Steps:

From	$eval : H \rightharpoonup H$	
To	$eval : (Item : H) \rightharpoonup H$	**Isomorphic to:**
	$eval : Item \to H \rightharpoonup H$	**which is unrolled to**
	$eval : Exp \to Flag \to BH \to CH \to H \rightharpoonup H$	

The pragmatic reason for unrolling is to make the first two parts e, α, of the beginning of the history visible, so they can be used for specialisation.[9]

7.2 Semantics 6

Domains:

$$\begin{array}{ll} \boxed{e \in Exp} & = \lambda - Expression \\ \boxed{\alpha \in Flag} & = \{T,\ F\} \\ h \in H, ch \in CH, bh \in BH & = [Item] \qquad (History) \\ it \in Item & = \langle Exp\ Flag\ BH\ CH \rangle \end{array}$$

Semantic functions:

$$\begin{array}{rcl} traversal & : & \boxed{Exp \rightharpoonup} H \\ \boxed{eval} & : & \boxed{Exp \to Flag \to} BH \to CH \to H \rightharpoonup H \\ \boxed{apk} & : & \boxed{Exp \to} CH \to H \rightharpoonup H \\ \boxed{evoperand} & : & Exp \to Flag \to H \to H \to H \rightharpoonup H \\ \boxed{lookup} & : & \boxed{Int \to Flag \to} BH \to CH \to H \rightharpoonup H \end{array}$$

Semantic equations :(M is the input λ-expression)

$$traversal = \boxed{eval\ M\ F}\,[\,]\ [\,]\ [\ \langle M\ F\ [\,]\ [\,]\rangle\ ]$$

$$\boxed{eval\ \ (FV\ \ x)\ \ \_}\ \ \_\ \ ch\ h\ \ = \boxed{\underline{apk\ (FV\ x)}}\ ch\ h$$

$$\boxed{eval\ (BV\ x\ i)\ \alpha}\ bh\ ch\ h\ \ = \boxed{\underline{lookup\ \ i\ \ \alpha}}\ bh\ ch\ h$$

$$\boxed{eval\quad \lambda x.e\quad T}\ bh\ ch\ h = \boxed{\underline{apk\ \ \lambda x.e}}\ ch\ h$$

$$\boxed{eval\quad \lambda x.e\quad F}\ bh\ ch\ h = \boxed{eval\ e\ \ F}\ bh\ ch\ \langle e\ \ F\ bh\ ch\rangle : h$$

$$\boxed{eval\ \ (e_1@e_2)\ \ \alpha}\ bh\ ch\ h = \boxed{eval\ e_1\ T}\ bh\ ch\ \langle e_1\ T\ bh\ ch\rangle : h$$

$$\boxed{apk\ \ \_}\ [\,]\qquad\qquad h = h$$

$$\boxed{apk\ \lambda x.e}\ (\langle \_\ \alpha\ bh\ ch'\rangle : \_)\ h\ \ = \boxed{\underline{eval\ e}}\ h\ ch'\ \langle e\ \alpha\ h\ ch'\rangle : h$$

$$\boxed{apk\ \ \_}\ (\langle e\ \alpha\ bh\ ch'\rangle : \_)\ h = \boxed{\underline{evoperand}}\ e\ F\ bh\ ch'\ h$$

$$\boxed{lookup\ 0\ \alpha}\ (\langle \_\ T\ \_\ ch'\rangle : \_)\ ch\ h = \textbf{case}\ ch'\ \textbf{of}$$
$$\langle e\ \_\ bh\ \_\rangle : \_ \Rightarrow \boxed{\underline{evoperand}}\ e\ \alpha\ bh\ ch\ h$$
$$\_ \qquad\qquad \Rightarrow$$
$$\textbf{case}\ ch\ \textbf{of}$$
$$[\,] \qquad\qquad \Rightarrow\ h$$
$$\langle ap\ \_\ bh''\ ch''\rangle : \_ \Rightarrow \boxed{\underline{evoperand}}\ ap\ F\ bh''\ ch''\rangle\ h$$

$$\boxed{lookup\ 0\ \_}\ (\langle \_\ F\ \_\ ch'\rangle : h')\ ch\ h = \boxed{\underline{apk\ (BV\ \_\ 0)}}\ ch\ h$$

$$\boxed{lookup\ i\ \alpha}\ (\langle \_\ \_\ bh'\ \_\rangle : \_)\ \ ch\ h\ = \boxed{lookup\ (i-1)\ \alpha}\ bh'\ ch\ h$$

[8] In some cases even more can be made static: we will see "The Trick" used to make static copies of dynamic values of bounded static variation, see discussion in [18].

[9] Remark: the *eval* argument is always a nonempty list, so case $H = [\,]$ need not be dealt with.

7.3 Need for and use of The Trick

The equations above lack one for $evoperand$. This presents a problem for specialisation: the item argument of $evoperand$ comes from a history, and histories are dynamic. Fortunately, however, the expression argument e in an item is BSV: of bounded static variation. In particular, it must be a subexpression of the original λ-expression M.

Let the applications present in M be $e_1^1@e_2^1, \ldots, e_1^m@e_2^m$. The following code does the trick. It is a sequence of tests on token values and branches.

$$\boxed{evoperand}\ e_1^1@e_2^1\ \ \alpha\ bh\ ch\ h = \boxed{\underline{eval\ e_2^1\ \alpha}}\ bh\ ch\ (\langle e_2^1\ \alpha\ bh\ ch\rangle : h)$$

$$\ldots$$

$$\boxed{evoperand}\ e_1^m@e_2^m\ \alpha\ bh\ ch\ h = \boxed{\underline{eval\ e_2^m\ \alpha}}\ bh\ ch\ (\langle e_2^m\ \alpha\ bh\ ch\rangle : h)$$

The residual program will contain one $evoperand$ definition for each application $e_1@e_2$ that is a subexpression of M. This code can be generated by the following annotated normaliser:

$$\begin{aligned}
&\boxed{evoperand}\ e\ \alpha\ bh\ ch\ h = \boxed{f\ M}\ e\ \textbf{where}\\
&\quad \boxed{f\ (e' : es)}\ e = \textbf{case}\ e'\ \textbf{of}\\
&\quad\quad \boxed{e_1@e_2} \Rightarrow \textbf{if}\ e = \boxed{e'}\ \textbf{then}\ \boxed{eval\ e_2\ \alpha}\ bh\ ch\ h\\
&\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else}\ \ \boxed{f\ es}\ e\\
&\quad\quad \boxed{_} \Rightarrow \boxed{f\ es}\ e
\end{aligned}$$

7.4 About the size of the compiled λ-expression

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, by using partial evaluation, the λ-free residual program UNP_M will have size $|UNP_M| = O(|M|)$. In words: M's LLL equivalent has *size that is only linearly larger* than M itself.

More generally: The size of specialised p_s will be linear in $|p|$ if for any p function $f(x_1, \ldots, x_n)$

- f has at most one static argument whose size depends on static input s; and
- each static argument is either completely bounded, or it is of BSV (bounded static variation).

This follows for UNP as a result of the binding time analysis described above.

8. Extension to a PCF-like language

Lambda-calculus is an elegant and simple Turing-complete language, but it is not really well-suited to program even simple functions. It turns out not to be hard to extend UNP to implement all the PCF operators. We give a glimpse here of how we extended traversal-based normalisation to handle lambda-calculi plus the control-flow operator *if-then-else*, the Y-combinator, numeric constants, and binary operations.

For the sake of brevity, we will only sketch a few parts of semantic equations that are related to new language primitives. Function apk is given additional arguments.

A quick overview, starting with the simplest case, a constant: call a continuation function.

Next consider the Y-combinator. When evaluated, it continues with evaluation of its body with an updated environment (binder pointer bh) without changing the current continuation ch. The $lookup$ function has to be extended: for function names it has to return the Y-combinator itself with an initial environment. These changes correspond strongly to the usual Y-combinator semantics $[\![Y\ M]\!] = [\![M(Y\ M)]\!]$.

Control-flow operators also fit into the traversal concept: evaluate the condition to be tested to some constant; and then decide, in the "apply continuation" function, which branch has to be chosen.

Finally, the case of operations on base values: For sake of simplicity, we only consider addition; other operations can be added in a similar way. There are different ways one can add addition to extend UNP. One way, is a syntactic extention. That is, addition can be seen as an alias for λ-term $\lambda m.\lambda n.\lambda x.(m@f)@((n@f)@x)$. This case nicely fits into the traversal concept since it is usual λ-term which can be evaluated by a complete head linear reduction.

On the other hand, this means that the results of computations will not be saved and every time the result needs to be used it will be recomputed. This is not usual for call-by-value arithmetic expression semantics which avoids multiple evaluation of the same arithmetic expression. Thus, the result of evaluation has to be stored somewhere. In order to be able to store an intermediate result some more machinery is needed: a token may be dynamic, i.e., a token may not be a subexpression of the input expression. Moreover, we allow a continuation function to dynamically change an argument of a binary operator. This makes it possible to store an intermediate results inside the traversal. Note that in this case the result is always a numerical constant.

```
eval h = let ⟨ e bh α ch ⟩ : _ = h in
  case e of
    ...
    Const n      ⇒ apk ch e h α
    Y  f  e1     ⇒ eval ⟨ e1  h α ch ⟩:h
    If  b e1 e2  ⇒ eval ⟨ b bh α ch ⟩:h
    Add  e1 e2   ⇒ eval ⟨ e1 bh T h ⟩:h


apk ch e h α = case ch of
  ...
  ⟨ (If  _  e1 e2) bh' α ch' ⟩ : _ ⇒
      if  e  == 0
      then eval ⟨ e2 bh' α ch' ⟩ : h
      else  eval ⟨ e1 bh' α ch' ⟩ :
  ⟨ (Add e1 e2) bh' α ch' ⟩ : ch'' ⇒
    let  Const n = e  in  if  α
      then eval ⟨ e2 bh' F ⟨ (Add e e2) bh' α ch') : ch'' ⟩ ⟩ : h
      else  let  Const n1 = e1
            in  eval ⟨ (Const (n1 + n)) bh' α ch' ⟩ : h
```

The result of traversing our code for function $sum(x) = \Sigma_{i=1}^{x} i$, i.e., when applied to constant 1

$$Y\ sum\ (\lambda x.\ \underline{if}\ x\ \underline{then}\ x\ +\ sum(x-1)\ \underline{else}\ 0)$$

is shown in Figure 4[10].

9. Conclusion

Time line of this research

Start: work on the simply-typed λ-calculus

[10] Note that $Y\ sum$ is **one** token which denotes a call of recursive function sum.

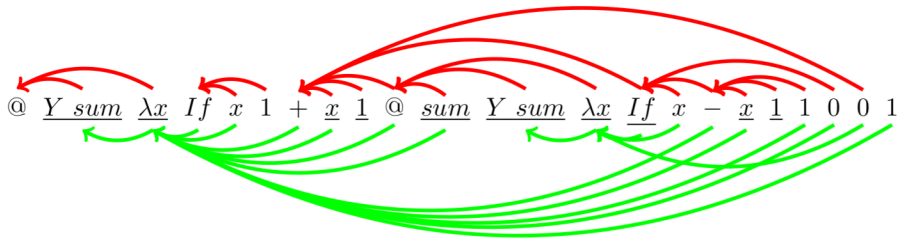


Figure 4: Traversal for $sum\ 1$.

We implemented one version of STNP in HASKELL and another in SCHEME. Experiments used the UNMIX partial evaluator (Romanenko [20]) to do automatic partial evaluation and compiler generation. A more complete HASKELL version included: typing; conversion to eta-long form; the traversal algorithm itself; and construction of the normalised term.

We also wrote STNP-gen (i.e., $[\![spec]\!](spec, UNP)$) by hand in SCHEME. This *generating extension* of STNP follows the lines of [21]; the same approach was used in Section 7. Effect: up $NP_M = [\![\text{STNP-gen}]\!](M)$. Program STNP-gen is essentially the compiler generated from STNP as in Section 6.2. STNP-gen yields output NP_M as a `scheme` program. This program has only surface differences from LLL as in Section 5.

Work on the untyped λ-calculus

An effective normaliser UNP was developed for ULC [22]. A traversal item has two back pointers (by comparison, STNP uses one). UNP was written in HASKELL and works on a variety of examples. A formal correctness proof of UNP is nearing completion, using the theorem prover COQ [22].

By specialising UNP, an arbitrary untyped λ-expression can be translated to LLL as in Section 5. A HASKELL generating extension has been written by hand.No SCHEME version or generating extension have yet been done, though this may be worthwhile for experiments using UNMIX.

Next steps

- More needs to be done to separate programs from data in ULC. (Appendix Section A is 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 likely relevant.
- It seems worthwhile to do *data-flow analysis* of output programs, e.g., for time and space optimisation of output programs.
- Using separated program and data naturally suggests that one investigate *computational complexity* phenomena (e.g., computational complexity of the λ-calculus).
- Yet another promising possibility is that LLL could serve as an intermediate language for a *semantics-directed compiler generator* [6].

10. Related works on the Lambda calculus

There are many works about λ-calculus evaluators that emphasise efficiency. Following is a brief overview of some of the most relevant approaches:

Many implementations of functional languages use lazy evaluation to ensure efficiency. Nevertheless, Wadsworth's original form of lazy evaluation [23] can fail to remove duplicate computations since it captures only values, but leaves functional terms intact. This is sometimes not considered a problem, because compilers [24] employ a fully lazy λ-lifting instead of Wadsworth's original transformation.

Sinot [25] extended Launchbury's [26] semantics for lazy evaluation. Sinot increased laziness by also sharing function bodies, using metavariables to represent open terms. It was also shown that complete lazy reduction can give exponential speedup in comparison with usual lazy evaluation [25].

Lévy [27] introduced the concept of *optimal reduction*. The main goal in an optimal reduction is to avoid redex duplications by adding some mechanisms to obtain redex sharing and *parallel reduction*. Moreover, Lévy showed that not only subexpressions, but closures as well, have to be shared to obtain optimality; and that among complete reductions, only lazy evaluation (call-by-need reductions) can optimallly reach the normal form.

In [28] Lamping presents a graph-based reducer that avoids duplication of work when possible. He represents a lambda term as a graph with special fan-in and fan-out nodes to model sharing; and defines a set of local graph rewriting rules. This scheme is shown to be Lévy-optimal by relating each beta reduction step to a Lévy parallel reduction step.

A similar approach by Shivers and Wand [29] represents λ-terms as directed acyclic graphs (DAGs) to allow sharing by means of back pointers and links. The back pointers are used to efficently search parents while links are used for efficent construction. The sharing that arises from β-reduction is efficiently managed by the DAG representation.

In [30] interaction nets are used as an intermediate representation for reduction. The name-free deBruijn λ-calculus [31] transforms λ-expressions into interaction nets [32] on which reduction together with so-called x-rules (rules that simplify interaction nets) are defined. Finally, a residual term is reconstructed from the resulting interaction net. The presented calculi are optimal in the sense of Lévy.

Lawall and Mairson [33] showed that the sharing graphs of Lamping and successors require exponentially many bookkeeping steps. One conclusion is a problem with the Lévy optimality criterion: achieving it (bookkeeping and all) would imply the collapse of some well-known complexity class containments.

References

[1] Gordon D. Plotkin. LCF considered as a programming language. Theor. Comput. Sci., 5(3):223255, 1977.

[2] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. Inf. Comput., 105(2):159267, 1993.

[3] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In Theoretical Aspects of Computer Software, International Conference TACS 94, pages 115, 1994.

[4] S. Abramsky and G. McCusker. Game semantics. In Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School, pages 156. Springer Verlag, 1999.

[5] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. Inf. Comput., 163(2):285408, 2000.

[6] Neil D. Jones, editor. Semantics-Directed Compiler Generation, volume 94 of Lecture Notes in Computer Science. Springer, 1980.

[7] David A. Schmidt. State transition machines for lambda calculus expressions. In Neil D. Jones, editor, Semantics-Directed Compiler Generation, volume 94 of Lecture Notes in Computer Science, pages 415440. Springer, 1980.

[8] C.-H. Luke Ong. Normalisation by traversals. CoRR, abs/1511.02629, 2015.

[9] 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), pages 8190, 2006.

[10] 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, ICFP12, pages 353364, 2012.

[11] Andrew D. Ker, Hanno Nickau, and C.-H. Luke Ong. Innocent game models of untyped lambda-calculus. Theor. Comput. Sci., 272(1-2):247292, 2002.

[12] Vincent Danos and Laurent Regnier. Head linear reduction. unpublished, 2004.

[13] Hendrik Pieter Barendregt. The lambda calculus : its syntax and semantics. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, New-York, Oxford, 1981.

[14] Peter Sestoft. Demonstrating lambda calculus reduction. In Torben . Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday], volume 2566 of Lecture Notes in Computer Science, pages 420435. Springer, 2002.

[15] Neil D. Jones. Transformation by interpreter specialisation. Sci. Comput. Program., 52:307339, 2004.

[16] John C. Reynolds. Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation, 11(4):363397, 1998 (originally 1972).

[17] Neil D. Jones. Computability and complexity - from a programming perspective. Foundations of computing series. MIT Press, 1997.

[18] 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.

[19] Richard Statman. The typed lambda-calculus is not elementary recursive. In 18th Annual Symposium on Foundations of Computer Science, pages 9094, 1977.

[20] S.A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Dines Bjrner, Andrei Ershov, and Neil D. Jones, editors, Partial Evaluation and Mixed Computation, pages 445463. North Holland, 1988.

[21] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M.V. Hermenegildo and J. Penjam, editors, Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP94, Madrid, Spain, September, 1994. (Lecture Notes in Computer Science, Vol. 844), pages 198214. Springer-Verlag, 1994.

[22] Daniil Berezun. UNP: strong normalisation of the untyped -expression by linear head reduction. ongoing work, 2016.

[23] Christopher P. Wadsworth. Semantics and Pragmatics of the Lambda-Calculus. PhD thesis, Oxford University, 1971.

[24] Simon L. Peyton Jones. The Implementation of Functional Programming Languages. Prentice-Hall, 1987.

[25] Franois-Rgis Sinot. Complete laziness: a natural semantics. Electr. Notes Theor. Comput. Sci., 204:129145, 2008.

[26] John Launchbury. A natural semantics for lazy evaluation. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 144154, 1993.

[27] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In To H.B. Curry: Essays on Combinatory Logic, Lambda Caclulus and Formalism. Academic Press, 1980.

[28] John Lamping. An algorithm for optimal lambda calculus reduction. In Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 90, pages 1630. ACM, 1990.

[29] Olin Shivers and Mitchell Wand. Bottom-up beta-reduction: Uplinks and lambda-dags. In Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, pages 217232, 2005.

[30] Kees-Lan can de Looij Vincent van Oostrom and Mariln Zwitserlood. Lambdascope another optimal implementation of the lambda-calculus. In Workshop on Algebra and Logic on Programming Systems (ALPS), 2004.

[31] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. INDAG. MATH, 34:381392, 1972.

[32] Yves Lafont. Interaction nets. In Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 90, pages 95108. ACM, 1990.

[33] Julia L. Lawall and Harry G. Mairson. Optimality and inefficiency: What isnt a cost model of the lambda calculus? In Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP 96), pages 92101, 1996.

[34] Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors. The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday], volume 2566 of Lecture Notes in Computer Science. Springer, 2002.

[35] William Blum and C.-H. Luke Ong. A concrete presentation of game semantics. In Galop 2008:Games for Logic and Programming Languages, 2008.

[36] William Blum and C.-H. Luke Ong. The safe lambda calculus. Logic Methods in Computer Science, 5(1), 2009.

[37] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girards execution formula). In Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS 93), pages 296306, 1993.

A. Should M have separate input data?

Motivations for giving M separate input data

In the current λ-calculus tradition M is self-contained; there is no dynamic data. The definition of a specialiser on a normaliser program UNP:

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

While this looks almost trivial, it is not. The left side describes a two-step process: first, M is translated into LLL (size linear in $|M|$). Then that program is run: without input, but perhaps giving a perhaps very long traversal.

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 UNP is extended to allow run-time input data.[11] The specialisation definition becomes:

$$\forall M \in \Lambda, d \in \mathit{Data} \,.\, [\![\, [\![spec]\!](\text{UNP}, M)]\!](d) = [\![\text{UNP}]\!](M, d)$$

Is adding separate input data a good idea? Let the specialiser output be $\text{UNP}_M = [\![spec]\!](\text{UNP}, M)$.

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

1. One motivation is that UNP$_M$ can be in a much simpler language than the λ-calculus. Our candidate: the "low-level language" LLL of Section 5.
2. A next step: it becomes natural to consider the *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.
3. Further, two stages are natural for semantics-directed compiler generation.

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 UNP$_{\texttt{mul}}$ is run. Expectations:

- Neither `mul` nor the data contain any loops or recursion. However `mul` will be compiled into an *LLL -program* UNP$_{\texttt{mul}}$ *with two nested loops*.
- Applied to two Church numerals m, n, UNP $_{\texttt{mul}}$ 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, and the pattern can be seen in Figure 1.)
- These appear as an artifact of the specialisation process. The reason the loops appear: While constructing UNP$_{\texttt{mul}}$ (i.e., during specialisation of UNP to its static input `mul`), the specialiser will encounter the same static values (subexpressions of M) more than once.