de Vrijer's Measure for SN of $\lambda^{\rightarrow}$ in Scheme

Nachum Dershowitz
Tel Aviv University, Israel

Abstract

We contribute a Scheme program for de Vrijer's proof of strong normalization of the simply-typed lambda calculus.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Type theory

Keywords and phrases Strong normalization, simply typed lambda calculus

Acknowledgements I thank the members of the lambda-calculus and rewriting group at Vrije University, Amsterdam, for the enjoyable meetings we had together shortly before the pandemic struck.

> Mr Howard, it shouldn't be too difficult to find a right ordinal assignment showing SN.
>
> Kurt Gödel to Bill Howard, as reported to me by Henk Barendregt

1 Gödel's Koan

Gödel's "koan" [4, 3, Problem 19], put forward by Jean-Jacques Lévy in 1991 and by others before and after (see [6, Problem 26]), asks for an easy, intuitive way to assign ordinals—be they natural numbers or transfinite ordinals—to terms of the simply-typed lambda calculus such that each (β-) reduction step of a term yields a smaller ordinal. There are a fair number of proofs of the strong normalization (SN), a.k.a. uniform termination, of (well-) typed lambda terms, including [9, 11, 5, 8, 7, 10], but none yet that meet the desideratum of intuitiveness. All the same, one compelling assignment that proves termination is that designed by Roel de Vrijer [2]. It measures a term by an overestimate of the maximum number of reduction steps, at the same time constructing a function for each term to provide that value when said term is applied to another.

I found it nontrivial to program de Vrijer's measure. Hence, this note. I follow a summary formulation by Henk Barendregt [1], and I program in Scheme (a variant of Lisp) for its built-in evaluation of lambda expressions. A lazy language might have been easier to program in.

2 de Vrijer's Measure

de Vrijer's measure $[\![e]\!]$ of typed term e generally consists of two components, what we call the "dot part" or "dot measure", $[\![e]\!]^\bullet$, and the "star part" or "star measure", $[\![e]\!]^*$. The star measure is an integer, an upper bound on the maximum length of a reduction sequence starting from e. But when calculating $[\![e]\!]$ recursively, one also needs to know what happens when e is a subterm embedded in a more complicated expression. The dot part serves this purpose. It is a function that knows how to compute the two-part measure $[\![et]\!]$ for any application of the current expression e to another expression t whose measure $[\![t]\!]$ is given. In other words, $[\![et]\!] = [\![e]\!]^\bullet([\![t]\!])$. The central consideration is that $mn + n + 1$ is an upper bound on the length of a reduction of a composition $(\lambda x.M)N$, where n bounds reductions of N and x appears at most m times in a reduct of M. So, each of the m copies of N can be reduced n times, plus one more step for the application of M to N, which first substitutes N for each (free) occurrence of x in the reduct. The additional term n in the (potentially over-) estimate is to cover the case when $m = 0$ but N is reduced nonetheless. The multiplier m is not computed in advance; rather, n gets plugged into M and tallied as its measure gets expanded. All this needs to be computed within an environment that tracks variable bindings dictated by the larger, enclosing term while building the measure recursively.

The whole measure can also be just a number, rather than a pair. In particular, the measure $[\![x]\!]$ of an unbound variable x is 0.

As a minimalist example, $[\![\lambda x^o.x]\!]$ (o is the base type) is essentially $\langle \lambda n.n+1, 0\rangle$. Since $\lambda x^o.x$ is in normal form, the star part $[\![\lambda x^o.x]\!]^*$ is 0. But if it's applied to a term with maximum reduction-sequence length n, then the full reduction will have an additional β step, so $[\![\lambda x^o.x]\!]^\bullet = \lambda n.n+1$.

Of course, one needs to prove that the measure in fact decreases with each β-step, regardless of the assignments (of measures) to free variables in the measure, for which see [2, Reduction Lemma, §3.5] or [1, Prop. 3.7].

de Vrijer [2, §4] also provides a more complicated, but precise, measure, which we do not address here.

The code is presented in the next section, followed by an example. Code and examples are made available at `http://nachum.org/SN.scm`.

3 My Scheme Code

3.1 Basics

A lambda term can be

(1) a typed variable, given as a variable symbol paired with a type expression, `(: x t)`,

(2) an application of one term to another, `(f t)`, or

(3) an abstraction (definition), `(lambda v t)`, comprising the atom λ, a bound typed variable (parameter) `v`, and a body `t`, which is itself a lambda term.

Types are either basic `o` or triples `(-> a b)`. Assignments are (association) lists of (dotted) pairs `(v . e)`, each giving a variable binding `v` $\mapsto$ `e`.

Accordingly, the basic constructors, destructors, and tests have the following simple definitions:

```
(define (atom? x) (not (list? x))) ; is-atom?
(define (var? x)
  (or (atom? x) (and (list? x) (equal? (car x) ':)))) ; is-variable?
(define (abs? x) (and (list? x) (equal? (car x) 'lambda))) ; is-abstraction?
(define (comp x y) (list x y)) ; application as list of two expressions
(define (fun x) (first x)) ; function part of application
(define (arg x) (second x)) ; argument part of application
(define (var x) (second x)) ; name of typed variable
(define (param x) (second (var x))) ; variable of abstraction
(define (kind x) (third (var x))) ; type of parameter
(define (body x) (third x)) ; body of abstraction
(define (mapsto x y) (cons x y)) ; binding as pair
(define (from y) (second y)) ; variable of binding
(define (to y) (third y)) ; assignment to variable
```

The following function extracts, or constructs, the type of a lambda expression, depending on its form:

```
(define (type x)
  (cond
    ((var? x) (third x))
    ((abs? x) `(-> ,(kind x) ,(type (body x))))
    (#t (third (type (fun x))))))
```

where ` is quote and , is unquote (evaluate first, before quoting). The case statement `cond` comprises a list of conditions plus values to return, and `#t` denotes the truth value, true.

We will use the form `(@ f t)`, which when executed forces the evaluation of `f` and `t` followed by application of the former to the latter:

```
(define (@ f t) ((eval f) (eval t))) ; apply!
```

where `eval` is a Scheme primitive.

3.2 The Two-Part Measure

The measure has two parts and will be represented by the expression `(L d s)`, where `d` is its dot part and `s`, the star part. Its constructors and destructors are

```
(define (L x y) `(L ,x ,y))
(define (dot x) (if (atom? x) x (second x))) ; dot part of measure
(define (star x) (if (atom? x) x (third x))) ; star part of measure
```

When the measure is a number, not a pair, both its parts are just that.

Given an expression `x` in an environment with assignments `v`, the following computes the measure by induction on the form of the expression:

```
(define (bra x v) ; compute measure
  (cond
    ((var? x) (value v x))
    ((abs? x)
     (let ((f (gensym)))
       (L `(lambda (,f)
             (add ,(bra (body x)
                        (cons (mapsto (var x) f) v))
                  (+ (star ,f) 1)))
          `(star ,(bra (body x)
                       (cons (mapsto (var x) (c (type (var x)) 0)) v))))))
    (#t `(@ (dot ,(bra (fun x) v)) ,(bra (arg x) v)))))
```

This is the heart of the method.

(1) For variables, one looks up what may already have been assigned to it, or else one creates a new initial valuation `c` that depends on the variable's type, as will be seen below.

(2) For abstractions, one constructs dot and star components. For its dot part, a new assignment is attached to the environment, the measure of the body is determined, and 1 is added for one beta-reduction step when it's applied, plus the bound on the steps needed to bring the function itself to normal form. The `gensym` command creates a new formal parameter for the dot measure, which is assigned in the environment to the variable of the abstraction that will also occur in the measure. (See below for the mechanism of addition.) Its star component is the star of the body with a promise for the assignment to the parameter.

(3) The last case is application of a function term to an argument term. This is exactly where the dot measure comes into play. The dot part of the pre-computed function's measure is applied to the argument's measure—whose evaluation has been delayed until now—to obtain the full measure for the combined application term.

The above `bra` function makes use of the following to create a new measure that increases both parts of the given measure `f` by a given integral amount `n`:

```
(define (add f n) ; add natural n to measure f
  (if (number? f)
      (+ f n)
      (let ((g (gensym)))
        (L `(lambda (,g) (add (@ (dot ,f) ,g) ,n))
           (+ (star f) n)))))
```

The star measure just gets added to, but the dot measure requires a new function that will do the addition when the time comes.

Applying an assignment is easy, treating the assignment as lookup in the association list of stored bindings:

```
(define (value v x) ; apply valuation v to variable x
  (if (assoc x v)
      (cdr (assoc x v))
      (c (type x) 0))) ; initial valuation
```

When necessary, viz. when the variable is not listed, this creates a new (typed) valuation `c`, using the following:

```
(define (c y n)
  (if (equal? y 'o) ; base type
      n
      (let ((f (gensym)))
        (L `(lambda (,f) (c (quote ,(to y)) (+ ,n (star ,f))))
            n))))
```

For a variable x of non-base type $\sigma \to \tau$, this valuation considers what happens when an instance of x is applied to some term f of type σ. The dot measure of x should return a valuation for (possibly compound) type τ that also incorporates the length of reductions given by the star measure $[\![f]\!]^*$.

Finally, to measure a top-level expression `x`, first construct and evaluate its measure in a pristine (empty) environment and then take its star part:

```
(define (o x) (star (eval (bra x '())))) ; the measure
```

4 Barendregt's Examples

I get the same values for all of Henk's examples [1] as he obtained manually, except for $(\lambda f^{o\to o}.\lambda x^{o}.fx)(\lambda x^{o\to o}.x)$, for which the above program calculates 2 as its star measure rather than 1, as therein. In our Scheme formalization this term is

```
(comp `(lambda ,f1 (lambda ,x0 ,(comp f1 x0))) `(lambda ,x1 ,x1))
```

The full, computed measure for this, just prior to a final evaluation-cum-application step `@`, looks like the following:

```
(@ (dot (L (lambda (g75)
             (add (L (lambda (g76)
                       (add (@ (dot g75) g76)
                            (+ (star g76) 1)))
                      (star (@ (dot g75) 0)))
                  (+ (star g75) 1)))
           (star (L (lambda (g78)
                      (add (@ (dot (L (lambda (g77)
                                        (c 'o (+ 0 (star g77))))
                                      0))
                               g78)
                           (+ (star g78) 1)))
                     (star (@ (dot (L (lambda (g77)
                                        (c 'o (+ 0 (star g77))))
                                      0))
                              0))))))
   (L (lambda (g79) (add g79 (+ (star g79) 1)))
      (star (L (lambda (g80) (c 'o (+ 0 (star g80))))
               0))))
```

Simplifying, this is equivalent to

```
(@ (lambda (g75)
     (add (L (lambda (g76)
               (add (@ (dot g75) g76)
                    (+ (star g76) 1)))
              (star (@ (dot g75) 0)))
          (+ (star g75) 1)))
   (L (lambda (g79) (add g79 (+ (star g79) 1)))
      0))
```

Evaluating and simplifying further yields the measure

```
(L (lambda (g76) (add (add (add g76 (+ (star g76) 1)) (+ (star g76) 1)) 1))
   2)
```

The dot part of this—when applied to a measure that is a plain number—is tantamount to $\lambda n.3n + 3$. Its star component is 2. Indeed, evaluating the expression takes two steps: $(\lambda f.\lambda x.fx)(\lambda x.x) \to_\beta \lambda x.(\lambda x.x)x \to_\beta \lambda x.x$. As Henk says [1]: "It isn't completely trivial to compute these: easy to make mistakes."

The truth is you don't really understand something until you've taught it to a computer, until you've been able to program it.

Don Knuth (2008)

References

1 Henk Barendregt. Digesting the proof of Roel de Vrijer that $\lambda_\rightarrow \models$ SN. Unpublished note, April 2019. URL: `http://nachum.org/Henk.pdf`.

2 Roel C. de Vrijer. Exactly estimating functionals and strong normalization. *Indagationes Mathematicae*, 49:479–493, 1987. URL: `https://core.ac.uk/reader/82154640`.

3 Nachum Dershowitz. The RTA list of open problems, 2009–2021. URL: `https://www.cs.tau.ac.il/~nachum/rtaloop/`.

4 Nachum Dershowitz, Jean-Pierre Jouannaud, and Jan Willem Klop. Open problems in rewriting. In R. Book, editor, *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications (Como, Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 445–456, Berlin, April 1991. Springer-Verlag. URL: `https://www.researchgate.net/publication/2441091`.

5 Robin O. Gandy. Proofs of strong normalization. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press Limited, 1980.

6 Ryu Hasegawa, Luca Paolini, and Paweł Urzyczyn. TLCA list of open problems, July 2014. URL: `http://tlca.di.unito.it/opltlca/`.

7 Assaf J. Kfoury and Joe B. Wells. New notions of reduction and non-semantic proofs of strong β-normalization in typed λ-calculi. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, pages 311–321, 1995. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.9949&rep=rep1&type=pdf`.

8 Jan Willem Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

9 Luis Elpidio Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, VIII(3):161–174, July 1967. `doi:10.1305/ndjfl/1093956080`.

10 Morten Heine Sørensen. Strong normalization from weak normalization in typed λ-calculi. *Information and Computation*, 133:35–71, 1997. `doi:10.1006/inco.1996.2622`.

11 William W. Tait. A realizability interpretation of the theory of species. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer, Boston, 1975.