Call-By-Value Lambda Calculus

Gert Smolka, Saarland University

November 22, 2017

We study a minimal functional programming language L realizing a call-by-value λ-calculus.

1 Introduction

We study a minimal functional programming language L realizing a call-by-value λ-calculus. L is an untyped language computing with procedures. In fact, procedures are the only values L computes with. No further values are needed since values of inductive data types can be represented as procedures.

From the perspective of full λ-calculus, L restricts β-reduction by disallowing reductions within abstractions and reductions where the argument term is not an abstraction. Moreover, L employs a simplified substitution operation that suffices for the reduction of closed terms.

Full λ-calculus is a deductive system rather than a programming language.

We assume the reader has an informal understanding of λ-terms, free and bound variables, substitution, and β-reduction. Readers who have worked with Coq will have this understanding.

We also assume that the reader has an informal understanding of the de Bruijn representation of terms.

2 De Bruijn Terms

We formalize **terms** with an inductive type using de Bruijn references:

$$s,t,u,v,w : \mathsf{Ter} \;::=\; n \mid st \mid \lambda s \qquad (n : \mathsf{N})$$

We call terms of the form n **variables**, terms of the form st **applications**, and terms of the form λs **abstractions**. We fix some terms for further use:

$\mathrm{I} = \lambda x.x$	$\mathrm{T} = \lambda xy.x$	$\mathrm{F} = \lambda xy.y$	$\omega = \lambda x.xx$	$\mathrm{D} = \lambda x.\omega\omega$
$:= \lambda 0$	$:= \lambda(\lambda 1)$	$:= \lambda(\lambda 0)$	$:= \lambda(00)$	$:= \lambda(\omega\omega)$

For readability, we will usually write terms with named abstractions, as shown above.

When we look at a term, we can distinguish between free and bound occurrences of variables. For instance, given the term $\lambda x.xy$, there is a bound occurrence of x and a free occurrence of y. We say that a variable x is **free** in a term s if x has a free occurrence in s. We formalize this notion with an inductive predicate free k s:

$$\frac{}{\text{free } k\ k} \qquad \frac{\text{free } k\ s \vee \text{free } k\ t}{\text{free } k\ (st)} \qquad \frac{\text{free } (Sk)\ s}{\text{free } k\ (\lambda s)}$$

We define a term s to be **closed** if there is no variable that is free in s. Moreover, we define a **procedure** to be a closed abstraction.

Fact 1 st is closed if and only if s and t are both closed.

Fact 2 A closed term is either an application or a procedure.

Exercise 3 Write a function that converts **Church terms**

$$M, N : \mathsf{CTer} \ ::= \ x \mid MN \mid \lambda x.M \qquad (x : \mathsf{N})$$

where abstractions introduce argument names into de Bruin terms as defined above. Write the function with an additional argument collecting the argument names of the traversed abstractions in a list. Convince yourself that the function can be used to obtain a decider for α-equivalence. Two terms are α-equivalent if they are equal up to consistent renaming of bound variables.

Exercise 4 Design and verify a decider for closedness of terms. One possibility is to write a boolean test that for s and k checks that all variables free in s are smaller than k.

3 Simple Substitution

We define a **substitution function** s^k_u that replaces every free occurrence of a variable k in a term s with a term u. The definition is by recursion on s:

$$\begin{aligned} n^k_u &= \text{if } n{=}k \text{ then } u \text{ else } n \\ (st)^k_u &= (s^k_u)(t^k_u) \\ (\lambda s)^k_u &= \lambda(s^{Sk}_u) \end{aligned}$$

Note that a substitution s^k_u may *capture* free variables in u; for instance, $(\lambda x.y)^y_x = \lambda x.x$. Note that capture does not happen if u is closed.

When we use substitution for formal reasoning, capturing must not occur. Our interest in the following is in computational evaluation of closed terms where capture cannot occur because of closedness.

Fact 5 If k is not free in s, then $s_t^k = s$.

Proof By induction on s. ∎

Fact 6 If s is closed, then $s_t^k = s$.

Proof Immediate consequence of Fact 5. ∎

Lemma 7 Let k be free in s_t^n and t be closed. Then $k \neq n$ and k is free in s.

Proof By induction on s. ∎

We fix the notation $\beta st := s_t^0$. The idea is that a β-redex $(\lambda s)t$ may be reduced to the term βst. Our definition of β is correct for such a β-reduction if λs and t are both closed.

Lemma 8 If λs and t are closed, then βst is closed.

Proof Follows with Lemma 7. ∎

4 Call-By-Value Reduction

The basic computation rule of λ-calculus is β-reduction: A term of the form $(\lambda x.s)t$ may be rewritten to the term s_t^x. For L we restrict β-reduction to subterms $(\lambda x.s)(\lambda y.t)$ that are not within an abstraction.

We formalize β-reduction for L with an inductive predicate $s \succ t$ defined as follows:

$$\frac{}{(\lambda s)(\lambda t) \succ \beta s(\lambda t)} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

We refer to the predicate $s \succ t$ as **one-step reduction**. One-step reduction is not functional since we have (II)(II) ≻ I(II), (II)(II) ≻ (II)I, and I(II) ≠ (II)I. Moreover, one-step reduction is not terminating since we have $\omega\omega \succ \omega\omega$.

Fact 9 If $s \succ t$ and s is closed, then t is closed.

Proof Induction on $s \succ t$ using Lemma 8. ∎

A term s is **reducible** if there exists a term t such that $s \succ t$.

Fact 10 A closed term is either a procedure or reducible.

Proof By induction on the term. ∎

Fact 11 If $s \succ^* s'$ and $t \succ^* t'$, then $st \succ^* s't'$.

Proof It suffices to show $st \succ^* s't$ and $s't \succ^* s't'$. The first claim follows by induction on $s \succ^* s'$, and the second claim follows by induction on $t \succ^* t'$. ∎

Exercise 12 Prove the following facts about reduction in L in Coq and make sure you understand the details.

a) $\omega\omega \succ \omega\omega$ and $\mathrm{D}(\lambda s) \succ \omega\omega$.

b) $\mathrm{F}(\lambda s)(\lambda t) \succ \mathrm{I}(\lambda t) \succ \lambda t$.

c) $\mathrm{T}(\lambda s)(\lambda t) \succ (\lambda(\lambda s))(\lambda t) \succ \lambda(s^1_{\lambda t})$.

Exercise 13 The following reductions are valid for all terms s and t in full λ-calculus. For each reduction, give minimal conditions for s and t making the reduction valid in L.

a) $\mathrm{I}s \succ s$ and $\mathrm{D}s \succ \omega\omega$.

b) $\mathrm{T}st \succ s$ and $\mathrm{F}st \succ t$.

Exercise 14 In full λ-calculus we have $(\lambda x.y)s \succ y$ if x and y are different variables. Explain why this is not true in L even if s is a procedure.

5 Call-By-Value Evaluation

We may reduce a term until we arrive at an irreducible term. Facts 10 and 9 tell us that such an iterated reduction process started with a closed term yields a procedure if it terminates. Seen abstractly, this yields an evaluation relation between closed term and procedures. Since one-step reduction is non-functional, we don't know whether the evaluation relation is functional. Moreover, we don't know whether the choice of the next reduction step affects termination. In fact, in full λ-calculus we have both $\mathrm{F}(\omega\omega) \succ \mathrm{F}(\omega\omega)$ and $\mathrm{F}(\omega\omega) \succ \mathrm{I}$. Note that $\mathrm{F}(\omega\omega) \succ \mathrm{I}$ does not hold for L.

We start our study of this situation by defining an inductive predicate $s \rhd t$:

$$\frac{}{\lambda s \rhd \lambda s} \qquad\qquad \frac{s \rhd \lambda u \qquad t \rhd v \qquad \beta u v \rhd w}{st \rhd w}$$

We read $s \rhd t$ as s **evaluates to** t. We say that a term s **evaluates** and write $\mathcal{E}s$ if $s \rhd t$ for some term t.

The evaluation predicate $s \rhd t$ provides a big-step semantics for L. We will see that this big-step semantics agrees the small-step semantics given by one-step reduction $s \succ t$.

Fact 15 $s \rhd t$ is functional.

Proof $s \rhd t_1 \to s \rhd t_2 \to t_1 = t_2$ follows by induction on $s \rhd t_1$. ∎

Fact 16 If $s \rhd t$, then t is an abstraction.

Proof By induction on $s \rhd t$. ∎

Fact 17 If $s \rhd t$ and s is closed, then t is a procedure.

Proof By induction on $s \rhd t$ using Lemma 8. ∎

Fact 18

1. Fst evaluates if and only if both s and t evaluate.
2. $\omega\omega$ does not evaluate.
3. Ds does not evaluate.

Proof For (2) one shows the implication $\omega\omega \rhd t \to \bot$ by induction on $\omega\omega \rhd t$. ∎

We now show that the small-step and big-step semantics for L agree. The proofs are essentially the same as the proofs we did for abstract expressions.

Fact 19 If $s \rhd t$, then $s \succ^* t$.

Proof By induction on $s \rhd t$ using Fact 11. ∎

Fact 20 If $s \succ s'$ and $s' \rhd t$, then $s \rhd t$.

Proof By induction on $s \succ s'$. ∎

Fact 21 If $s \succ^* t$ and t is an abstraction, then $s \rhd t$.

Proof By induction on $s \succ^* t$ using Fact 20. ∎

Corollary 22 Let $s \succ^* t_1$ and $s \succ^* t_2$, and let t_1 and t_2 be abstractions. Then $t_1 = t_2$.

Proof Follows with Facts 20 and 15. ∎

We remark that the agreement between small-step and big-step semantics can be shown for every substitution function s_t^x. The reason is that the proofs of the relevant facts (11, 19, 20, 21) do not depend on properties of the substitution function.

Exercise 23 Show that Tst evaluates if s and t evaluate. Note that the proof is complicated by the fact that we use capturing substitution.

6 Uniform Confluence

Fact 24 (Uniform confluence) Let $s \succ t_1$ and $s \succ t_2$. Then either $t_1 = t_2$ or $t_1 \succ u$ and $t_2 \succ u$ for some u.

Proof By induction on $s \succ t_1$.

1. Let $s = (\lambda s_1)(\lambda s_2)$. Then $t_1 = t_2$ since abstractions are irreducible.
2. Let $s = s_1 s_2$, $s_1 \succ s_1'$, and $t_1 = s_1' s_2$. Case analysis on $s \succ t_2$.
   a) $s_1 \succ s_1''$ and $t_2 = s_1'' s_2$. The claim follows with the inductive hypothesis for $s_1 \succ s_1'$.
   b) $s_2 \succ s_2'$, and $t_2 = s_1 s_2'$. The claim follows with $u = s_1' s_2'$.
3. Analogous to (2). ∎

The intuitive reason reduction in L is uniformly confluent is that only outermost β-redexes can be reduced and the reduction of a β-redex is functional (the β function). Hence, if $s \succ t_1$ and $s \succ t_2$, either the same redex is reduced and thus $t_1 = t_2$, or two disjoint redexes u_1 and u_2 are reduced and thus t_1 and t_2 can be joined by reducing the remaining redex u_2 in t_1 and the remaining redex u_1 in t_2.

As can be shown generally for abstraction reduction systems, uniform confluence has the important consequence that every weakly normalising term is strongly normalising. Moreover, if $s \succ^n t$ and t is irreducible, the length of every reduction chain issuing from s is bound by n.

7 Step-Indexed Evaluator

The evaluation predicate $s \rhd t$ is functional and partial. We cannot expect that we can define a total function $f : \mathsf{Ter} \to \mathsf{O}(\mathsf{Ter})$ such that $fs = \lfloor t \rfloor \leftrightarrow s \rhd t$ (since existence of such a function implies decidability of the halting problem for L). However, we can define a function $E : \mathsf{N} \to \mathsf{Ter} \to \mathsf{O}(\mathsf{Ter})$ such that

$$s \rhd t \;\leftrightarrow\; \exists n.\, Ens = \lfloor t \rfloor$$

We call such a function a **step-indexed evaluator**. The step index limits the recursion depth of the evaluator, which provides for termination.

We define a step-indexed evaluator E satisfying the following equations:

$$\begin{aligned} E\ n\ k &= \emptyset \\ E\ n\ (\lambda s) &= \lfloor \lambda s \rfloor \\ E\ 0\ (st) &= \emptyset \\ E\ (Sn)\ (st) &= \mathsf{match}\ E\ n\ s,\ E\ n\ t\ \mathsf{with} \\ &\quad \mid \lfloor \lambda s \rfloor,\ \lfloor t \rfloor \Rightarrow E\ n\ s_t^0 \\ &\quad \mid \_\ \_ \Rightarrow \emptyset \end{aligned}$$

Note that the formal definition of E will have 5 matches.

We show the correctness of E with respect to the evaluation predicate $s \triangleright t$.

Fact 25

1. If $E\,0\,s = \lfloor t \rfloor$, then s is an abstraction and $t = s$.
2. If $E\,n\,s = \lfloor t \rfloor$, then $E\,(Sn)\,s = \lfloor t \rfloor$.
3. If $E\,m\,s = \lfloor t \rfloor$ and $m \le n$, then $E\,n\,s = \lfloor t \rfloor$.
4. If $s \triangleright t$, then $E\,n\,s = \lfloor t \rfloor$ for some n.
5. If $E\,n\,s = \lfloor t \rfloor$, then $s \triangleright t$.

Proof Claim 1 follows by case analysis on s. Claim 2 follows by induction on n using (1). Claim 3 follows by induction on $m \le n$ using (2). Claim 4 follows by induction on $s \triangleright t$ using (3). Claim 5 follows by induction on n using (1). The formal proofs of (2), (4), and (5) are clumsy due to the 5 matches in E. ∎

Theorem 26 (Agreement) $s \triangleright t \leftrightarrow \exists n.\, Ens = \lfloor t \rfloor$.

Proof Immediate with Fact 25. ∎

We call a predicate $p : X \to \mathbf{P}$ **modest** if there is a function $(\exists x.\, px) \to (\Sigma x.\, px)$. A basic result about Coq's type theory says that decidable predicates on countable types are modest (constructive choice). Using the step-indexed evaluator, we now show that $\lambda t.\, s \triangleright t$ is modest for every s. In other words, there is a function that given a term s and a proof that s evaluates yields a term t such that $s \triangleright t$.

Theorem 27 (Modesty of Evaluation) There is a function $\forall s.\, \mathcal{E}s \to \Sigma t.\, s \triangleright t$.

Proof Let $\mathcal{E}s$. Then the predicate $\lambda n. \exists t.\, Ens = \lfloor t \rfloor$ is decidable and satisfiable. Hence constructive choice fo N gives us an n such that $\exists t.\, Ens = \lfloor t \rfloor$. By case analysis on the option Ens we obtain t with $Ens = \lfloor t \rfloor$. Now $s \triangleright t$ by Theorem 26. ∎

Exercise 28 (Proof by computation) Prove the following propositions in Coq:

a) $(\mathrm{T}(\mathrm{F}\,\mathrm{D}\,\mathrm{I})\mathrm{D})\,\mathrm{I} \triangleright \mathrm{I}$

b) $((\mathrm{T}(\mathrm{F}\,\mathrm{D}\,\mathrm{I})\mathrm{D})\,\mathrm{I})\,((\mathrm{T}(\mathrm{F}\,\mathrm{D}\,\mathrm{I})\mathrm{D})\,\mathrm{I}) \triangleright \mathrm{I}$

Note that straightforward proofs using Theorem 26 exist since verifications $Ens = \lfloor t \rfloor$ can be done by computation for concrete n, s, and t.

8 Scott Encoding of Numbers

Values of inductive data types can be represented as procedures following a scheme due to Dana Scott.

Consider the inductive data type for natural numbers in Coq:

$$\mathsf{nat} \;:=\; O : \mathsf{nat} \mid S : \mathsf{nat} \to \mathsf{nat}$$

The type provides us with two constructors O and S and a match for numbers. We may represent the match for numbers with the function

$$Mnab \;:=\; \mathsf{match}\ n \mid O \Rightarrow a \mid Sn' \Rightarrow bn'$$

The function represents the two rules of the match with the arguments a and b. We call these arguments **continuations**. We have one continuation per constructor since a basic match has one rule per constructor.

Following this idea, we represent a number n in L as a procedure computing the function Mn. This gives us the following representation for numbers:

$$\begin{aligned} \hat{0} &:= \lambda ab.a \\ \widehat{Sn} &:= \lambda ab.b\,\hat{n} \end{aligned}$$

Fact 29 Let n be a number and u and v be procedures.
Then $\hat{0}\,uv \succ^* u$ and $\widehat{Sn}\,uv \succ^* v\,\hat{n}$.

Fact 30 For all numbers m and n, the terms $\widehat{m}$ and $\hat{n}$ are procedures such that $\widehat{m} = \hat{n}$ if and only if $m = n$.

We call the procedures $\hat{n}$ **Scott numerals**. We define procedures computing successors and predecessors of Scott numerals:

$$\begin{aligned} \mathsf{Succ} &:= \lambda x.\,\lambda ab.bx \\ \mathsf{Pred} &:= \lambda x.\,x\,\hat{0}\,\mathrm{I} \end{aligned}$$

Verifying the correctness of Succ and Pred is straightforward.

Fact 31 $\mathsf{Succ}\,\hat{n} \succ^* \widehat{Sn}$, $\mathsf{Pred}\,\hat{0} \succ^* \hat{0}$, $\mathsf{Pred}\,\widehat{Sn} \succ^* \hat{n}$.

Exercise 32 Write procedures P, π_1, and π_2 such that $\pi_1(Pu_1u_2) \succ^* u_1$ and $\pi_2(Pu_1u_2) \succ^* u_2$ for all procedures u_1 and u_2.

Exercise 33 Recall that terms are an inductive datatype with three constructors. Give the Scott encodings of the terms k, st, and λs.

9 Recursion Operator

L can express unguarded recursion. The idea is to obtain a recursive procedure from a non-recursive procedure serving as *template* for the recursive procedure. Given a recursive procedure, a template for the procedure can be obtained by taking the recursive procedure as argument. Here is a template for a recursive addition function in Coq:

$$\lambda f x y.\ \mathsf{match}\ x\ [0 \Rightarrow y \mid Sx' \Rightarrow S(fx'y)]$$

Note that in Coq we can translate a template into the accompanying recursive function by replacing the λ with fix (the symbol for recursive abstractions). Once we write fix in place of λ, Coq will check that the recursion is structural.

For L we can define a function $\rho : \mathsf{Ter} \to \mathsf{Ter}$ that for every template procedure yields a procedure that simulates the recursive procedure described by the template. We call ρ a recursion operator for L. The recursion operator will transform every template procedure into a quasi-recursive procedure without checking whether the recursion expressed by the template is structural.

We may say that ρ yields for a template u a procedure that simulates the recursion described by u. As is well-known from the full λ-calculus, recursion can be simulated with self-application (as present in $\lambda x.xx$).

It turns out that there is an elegant formal specification for recursion operators. Thus when we prove properties of quasi-recursive procedures obtained with a recursion operator, there is no need to know the somewhat technical definition of the recursion operator.

Fact 34 (Recursion Operator) There is a function ρ from terms to terms such that ρs is a procedure if s is closed and

$$(\rho u)v \ \succ^3\ u(\rho u)v$$

for all procedures u and v.

Proof $\rho s := \lambda x.CCsx$ with $C := \lambda xy.y(\lambda z.xxyz)$ does the job. ∎

Exercise 35 Consider in Coq the recursive function

$$\mathsf{double} := \mathsf{fix}\ fx.\ \mathsf{match}\ x\ [0 \Rightarrow 0 \mid Sx' \Rightarrow S(S(fx'))]$$

a) Write the template function Double for double.

b) Prove Double double x = double x.

c) Prove $(\forall x.\ \mathsf{Double}\ f\ x = fx) \to fx = \mathsf{double}\, x$.

Exercise 36 Carefully verify the reduction $(\rho u)v \ \succ^3\ u(\rho u)v$ from Fact 34.

10 Case Study: Addition in L

Programming in L is convenient since we can follow familiar patterns from functional programming. We demonstrate the case with a *functional specification*

$$\forall mn.\ \text{add}\ \widehat{m}\ \widehat{n} \succ^* \widehat{m+n}$$

of a procedure add for addition. We say that we are looking for a procedure add *realising* the addition function $m+n$. A well-known *recursive specification* for an addition function consists of the equations

$$0 + n = n$$
$$Sm + n = S(m+n)$$

Following the recursive specification, we can realize an addition function in Coq with a recursive abstraction:

$$\mathsf{fix}\ fmn.\ \mathsf{match}\ [0 \Rightarrow n \mid Sm' \Rightarrow S(fm'n)]$$

Using the recursion operator and Scott numerals, we translate the recursive function into a procedure in L:

$$\text{Add} := \lambda fxy.\ xy(\lambda x'.\ \text{Succ}(fx'y))$$
$$\text{add} := \rho\ \text{Add}$$

Fact 37 $\text{add}\ \widehat{m}\ \widehat{n} \succ^* \widehat{m+n}$.

Proof By induction on m using Fact 34. ∎

The functional specification of add has the virtue that properties of add like commutativity follow immediately from properties of the addition function $m+n$.

Exercise 38 Prove $\text{add}\ \widehat{m}\ \widehat{n} \succ^* \widehat{n+m}$ using Fact 37.

Exercise 39 Verify the following reductions for all procedures u, v, w and all numbers m, n:

a) $\text{Add}\ w\ u\ v \succ^* u\ v\ (\lambda x.\ \text{Succ}\,(wxv))$.

b) $\text{add}\ \widehat{Sm}\ \widehat{n} \succ^* \text{Succ}\,(\text{add}\ \widehat{m}\ \widehat{n})$.

Exercise 40 Realize and verify a procedure double following the recursive specification in Exercise 35. Do not use the procedure add.

Exercise 41 Realize and verify a procedure for multiplication of Scott numerals. Use the procedure add and the correctness result shown for add.

11 Discussion

It is very important that one can verify reductions like add $\widehat{m}\ \widehat{n} \succ^* \widehat{m+n}$ by hand. It is a remarkable feature of the λ-calculus that hand verification of such reductions is straightforward. Use the accompanying Coq development to simulate hand verifications with tactics.

The call-by-value calculus presented here is from Forster and Smolka [1]. There you will find references to the relevant literature.

A call-by-value λ-calculus was first studied in the 1970s by Gordon Plotkin [2]. Plotkin also reduces terms of the form $(\lambda x.s)y$ where the argument term is a variable. Moreover, Plotkin defines one-step reduction deterministically by choosing the left-most redex. Plotkin works with Church terms and non-capturing substitution renaming bound variables.

References

[1] Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017, Brasília, Brazil*, volume 10499 of *LNCS*, pages 189–206. Springer, 2017.

[2] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.