SAARLAND UNIVERSITY
FACULTY OF NATURAL SCIENCES AND TECHNOLOGY I

BACHELOR'S THESIS

Verified Compilation of Weak Call-by-Value Lambda-Calculus into Combinators and Closures

Fabian Maximilian Kunze

Advisor:
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Markus Bläser

Submitted on November 27, 2015

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath:

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent:

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 27th November, 2015

Abstract

In this thesis, we relate the weak call-by-value λ-calculus L to two other systems: a call-by-value combinatory logic called SKv and a variant of L with closures, called LC. All proofs are carried out in the proof assistant Coq.

We show that L can be fully embedded into SKv by giving an injective mapping from L to SKv that preserves term equivalence and irreducibility in both directions.

We show that LC refines the semantics of L: It is sound and complete with respect to L, while allowing to postpone certain parts of computations until they become necessary.

We give a sound and complete interpreter for LC that can be used to interpret L.

Acknowledgements

First I would like to express my gratitude to Prof. Smolka for the guidance during numerous meetings. His support was far above what can be expected. The other people of the chair were always helpful as well.

I would also like to thank my friends and family for supporting and motivating me during the writing of this thesis. Especially Yannick, Noemi and Clara for taking time to proofread parts of this thesis.

Finally, I would like to thank Loscher for producing Club Mate.

Contents

Abstract **v**

1 Introduction **1**

2 Uniform Confluence **5**

3 L: Weak Call-by-Value λ-Calculus **9**
3.1 L as Programming Language 11
3.2 Semi-Automated Internalization 12
3.3 Named L 12

4 SKv: Call-by-Value Combinatory Logic **15**
4.1 Miscellaneous 17

5 Embedding L into SKv **19**
5.1 Pseudo-Abstraction 19
5.2 Compiling L into SKv 21
5.3 $\mathcal{C}$ is invertible 22
5.4 $\mathcal{C}$ is Complete 23
5.5 Conclusion 25

6 LC: L with Closures **27**
6.1 Drawbacks of Substitution Semantics in L 27
6.2 Basic Properties of LC 28
6.3 Connecting LC and L 29
6.4 LC is Complete 32
6.5 Evaluating LC 34
6.6 Related Systems 35

7 Coq Formalization **37**

Chapter 1

Introduction

Since introduced by Church in the 1930s, λ-calculus had a great impact on the fields of computability theory, programming languages and logic [Bar97] [Bar84].

The terms of λ-calculus represent higher order functions and are composed from three primitives: variables, abstractions and applications. Other datatypes and their operations can be encoded as functions.

The different variants of λ-calculus form the base for functional programming languages that are in practical use today, like OCaml or Haskell. In this thesis, we will mostly refer to $\lambda\beta$, untyped λ-calculus with β-reduction.

Compared with Turing machines, which define the same notion of computability, $\lambda\beta$ has an applicative structure that directly allows to compose two terms, but a fair bit of work is needed to compose two Turing machines [AR15]. This allows for easier formal reasoning using $\lambda\beta$, whereby using Turing machines, it is common practice to define an algorithm in terms of pseudocode or a plain text description and not with giving a concrete Turing machine.

The benefit of Turing machines is that the inherent notion of a primitive step directly suggest a reasonable cost model for the time that is required by a computation. This forms the base for the field of complexity theory. In contrast to that, the number of primitive steps for a computation in $\lambda\beta$ largely depends on the specific reduction strategy.

Weak Call-by-Value λ-Calculus

The ambiguity in the length of a computation can be resolved by restricting $\lambda\beta$ to weak call-by-value reductions [LM08]. In this system, the number of primitive steps in a computation is independent of the reduction strategy. As we will see in this thesis, this is a result of the *uniform confluence* property, and strongly connected to the call-by-value aspect of the reduction relation.

Weak call-by-value λ-calculus still is Turing complete, and a lot of programming techniques used in $\lambda\beta$ or functional programming languages also

apply to it. But the semantics is considerably simpler since the weak aspect of the reduction strategy allows for a straightforward notion of substitution without the need for renaming variables. The weak call-by-value reduction relation simplifies the proof of basic properties like confluence compared to untyped λ-calculus. Basic results of computability theory are formalized for L, a concrete weak call-by-value λ-calculus, in [For14]. An approach to formal complexity theory using weak call-by-value λ-calculus is shown in [LM08] and [Rot15].

The interest in the weak call-by-value λ-calculus motivates to transfer results and ideas from $\lambda\beta$. One is the relation between $\lambda\beta$ and the combinatory logic SKI, another formal system build around higher order functions. SKI does not allow the definition of arbitrary functions, but only three build-in functions called combinators, from which new functions can be derived via composition.

Combinatory logic was introduced in 1920 by Schönfinkel to express first order logic without the need for bound variables [Sch24]. Later, it was discovered that the combinatory logic SKI can also be used as a computational system with capabilities similar to $\lambda\beta$.

Despite the seemingly severe restriction to only three build-in functions, SKI equipped with an extensional notion of term equivalence nearly incorporates the full semantics of $\lambda\beta$ [HS08]. This shows that the ability to define arbitrary functions via abstractions is not required, but just is convenient for a programming system based on functional application.

But the relation between $\lambda\beta$ and SKI is not fully satisfying as an altered notion of term equivalence in SKI is required [HS08].

Contributions

The main contribution of this thesis is the verified compilation of L into two other formal systems.

Call-by-Value Combinator Logic

In this thesis, we define the call-by-value combinatory logic SKv. We fully embed L into SKv, that is, we give a left-invertible mapping from L to SKv such that irreducibility and term equivalence are preserved in both directions. Since we do not need an altered notion of reducibility and term equivalence for this, the correspondence between L and SKv is more satisfying than the one between $\lambda\beta$ and SKI.

Weak Call-by-Value λ-Calculus with Closures

We also approach a more practical problem: the efficient evaluation of L. For this, we introduce LC, a variant of L with closures. Using ideas also appearing in the definition of Standart ML [MTHM97], the semantics of

LC is not defined using substitution. Instead, we annotate L-terms with an environment. Arguments of abstractions are inserted to the environment and the value of a bound variable is looked up in the environment when needed.

The motivation introducing LC is the refinement of L with respect to substitution. The substitution performed in a β-reduction in L corresponds to several, smaller reduction steps in LC, which are only carried out as far as needed to proceed with the reduction of the term. Therefore, substitutions not affecting the result of the evaluation in L can be avoided in LC.

Coq

All our results are formalized in the proof assistant Coq. Coq is an implementation of the calculus of inductive constructions (CIC), a constructive type theory. Interestingly, this system ultimately originates from the second use of λ-calculus intended by Church, as a formal logic.

In Coq, each proposition is a type, and a proof of a proposition is a well-typed expression of that type. So programming and proving in Coq can be done in the same system.

The introduction of LC was motivated by the Coq formalization of [For14]. Since reasoning about a programming language often requires the evaluation of terms, an efficient evaluation was needed to speed up the compilation of the proofs.

Overview

In the first two chapters, we summarise some results that can be found in the literature: Chapter 2 contains properties related to uniform confluence and Chapter 3 gives the definition and basic properties of L.

In Chapter 4, we define the call-by-value combinatory logic SKv. We also prove some basic properties, like uniform confluence.

We relate L and SKv in Chapter 5 and show that L can be fully embedded in L. We give a mapping from L to SKv, show that this mapping is compatible with term equivalence and irreducibility in both directions, and give a left-inverse allowing to pull back reductions from SKv to L.

In Chapter 6, we define a variant of L with closures: LC. We prove that the reduction relation in LC refines L-reduction, give an sound and complete interpreter for LC and show how this can be used to evaluate L-terms.

An overview of the accompanying Coq formalization can be found in Chapter 7.

Chapter 2

Uniform Confluence

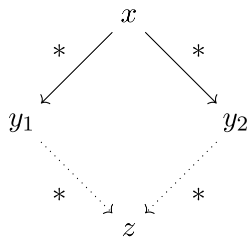
Throughout this thesis, we work with different reduction systems. They all are *uniformly confluent* [Nie00], a property implying that the reduction strategy does not influence the length of normalizing reduction chains.

Since we did not find this and related properties in a condensed and abstract form in the literature, we will do so here.

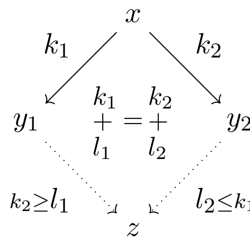
For this chapter we assume an abstract reduction system consisting of a set X and a relation $\rightarrow \subseteq X \times X$. We use the notations from [BN98]. By x, y and z we will denote elements of X, while k, l and m range over the natural numbers.

Recall that $\rightarrow$ is **confluent** iff from $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2$ we can conclude that there exists z with $y_1 \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y_2$.

Definition 2.1 $\rightarrow$ is **uniformly confluent** if from $y_1 \overset{k_1}{\leftarrow} x \overset{k_2}{\rightarrow} y_2$ we can conclude that there exists z, $l_1 \leq k_2$, and $l_2 \leq k_1$ such that $y_1 \overset{l_1}{\rightarrow} z \overset{l_2}{\leftarrow} y_2$ with $k_1 + l_1 = k_2 + l_2$.



(a) Confluence



(b) Uniform Confluence

Figure 2.1: Confluences

As seen in Figure 2.1, this refines confluence. Note that in all diagrams in this chapter, the stated side conditions require the length of reductions to be uniform over different paths.

Instantiating Definition 2.1 with $k_1 = k_2 = 1$, we conclude a more local property implied by uniform confluence:

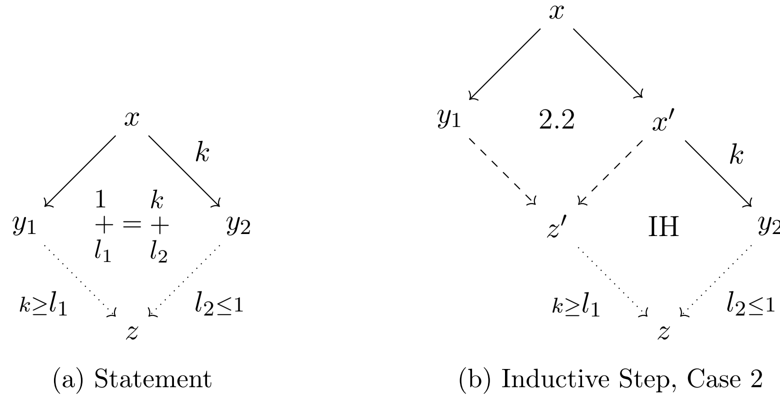
Definition 2.2 $\rightarrow$ has the **uniform diamond property** if from $y_1 \leftarrow x \rightarrow y_2$ we can conclude that either $y_1 = y_2$ or that there exists z with $y_1 \rightarrow z \leftarrow y_2$.

This property was first mentioned in [New42]. In [Nie00], it is introduced under the name 'uniform confluence'. We reserve this notion for an equivalent, but more directly applicable characterization.

In this thesis, all reduction relations turn out to have the uniform diamond property by similar proofs: We only look at call-by-value term rewriting systems and in those, two redexes in the same term either do not overlap or are identical. In the first case, we can join the two terms by reducing the other redex, in the later case, the redexes contract to the same term.

We will now show that the uniform diamond property already implies uniform confluence. This is shown for a specific relation in [LM08]. We essentially generalize this proof and follow the idea in [For15].

Lemma 2.3 (Uniform Semi-Confluence) Assume $\rightarrow$ has the uniform diamond property and $y_1 \leftarrow x \stackrel{k}{\rightarrow} y_2$. Then there exist z, $l_1 \leq k$ and $l_2 \leq 1$ with $y_1 \stackrel{l_1}{\rightarrow} z \stackrel{l_2}{\leftarrow} y_2$ such that $1 + l_1 = k + l_2$.



(a) Statement

(b) Inductive Step, Case 2

Figure 2.2: Lemma 2.3

Proof By induction on k. For $k = 0$, we note that y_1 has the desired properties. Otherwise, we have $x \stackrel{1+k}{\rightarrow} y_2$, so there is x' with $x \rightarrow x' \stackrel{k}{\rightarrow} y_2$. We use the uniform diamond property with $y_1 \leftarrow x \rightarrow x'$ and get two cases:

- If $y_1 = y_2$, then the claim follows for $z = y_2$.

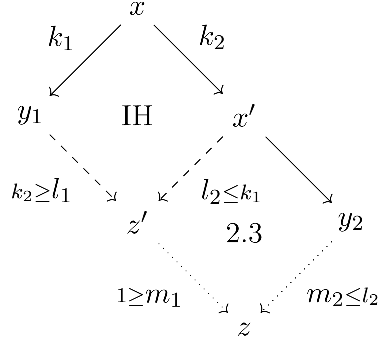


Figure 2.3: Recursive Step Theorem 2.4

- Otherwise there is z' with $y_1 \to z' \leftarrow x'$. Then (see Figure 2.2b) the induction hypothesis applied to $z' \leftarrow x' \overset{k}{\to} y_2$ yields a term z and two numbers $l_1 \leq k$ and $l_2 \leq 1$ such that $z' \overset{l_1}{\to} z \overset{l_2}{\leftarrow} y_2$ with $1 + l_1 = k + l_2$. z has the desired properties. ■

Theorem 2.4 A relation is uniformly confluent iff it has the uniform diamond property.

Proof Any uniformly confluent relation has the uniform diamond property by instantiating $k_1 = k_2 = 1$ in Definition 2.1.

For the converse direction, assume that $\to$ has the uniform diamond property and $y_1 \overset{k_1}{\leftarrow} x \overset{k_2}{\to} y_2$. We show by induction on k_2 that there are z, $m_1 \leq k_2$ and $m_2 \leq k_1$ such that $y_1 \overset{m_1}{\to} z \overset{m_2}{\leftarrow} y_2$ with $k_1 + m_1 = k_2 + m_2$.

For $k_2 = 0$, the claim follows for $z = y_1$. Otherwise, we know that there is x' with $x \overset{k_2}{\to} x' \to y_2$ (see Figure 2.3). By our induction hypothesis used with $y_1 \overset{k_1}{\leftarrow} x \overset{k_2}{\to} y_2$, there are $l_1 \leq k_2$ and $l_2 \leq k_1$ and a term z' such that $y_1 \overset{l_1}{\to} z' \overset{l_2}{\leftarrow} x'$ with $k_1 + l_2 = k_2 + l_1$.

Using Lemma 2.3 on $z' \overset{l_2}{\leftarrow} x' \to y_2$ yields $m_1 \leq 1$ and $m_2 \leq l_2$ and a term z such that $z' \overset{m_1}{\to} z \overset{m_2}{\leftarrow} y_2$ with $l_2 + m_1 = 1 + m_2$. The claim follows with z.■

Note that *uniform* confluence implies the *uniform* diamond property, but confluence does not imply the diamond property [BN98]. By arithmetic reasoning, we can also derive the characterization of uniform confluence (see Figure 2.4) mentioned in the proof of Proposition 2.4 in [Nie00], .

Corollary 2.5 A relation is uniformly confluent iff from $y_1 \overset{k_1}{\leftarrow} x \overset{k_2}{\to} y_2$, we can conclude that there exists z and $m \leq \min(k_1, k_2)$ such that $y_1 \overset{k_2-m}{\to} z \overset{k_1-m}{\leftarrow} y_2$.

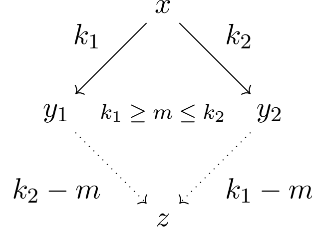


Figure 2.4: Alternative Characterization of Uniform Confluence

In our concrete systems, m corresponds to the number of shared reduction steps in the two derivations $y_1 \stackrel{k_1}{\leftarrow} x \stackrel{k_2}{\rightarrow} y_2$.

Proof Assuming uniform confluence, the other characterization follows from arithmetic reasoning taking $m = (k_1 + k_2) - (k_1 + l_1)$. In the other direction, arithmetic reasoning taking $l_1 = k_2 - m$ and $l_2 = k_1 - m$ yields uniform confluence. ■

We now derive some properties from uniform confluence. Recall that a relation $\rightarrow$ is **Church-Rosser** iff from $x_1 \stackrel{*}{\leftrightarrow} x_2$ we can conclude that there exists y with $x_1 \stackrel{*}{\rightarrow} y \stackrel{*}{\leftarrow} x_2$. As proven in [BN98], the Church-Rosser property is equivalent to confluence.

Corollary 2.6 Every uniformly confluent relation is Church-Rosser.

Another relevant consequence was mentioned in [Nie00]. Recall that a term x is **irreducible** (or **normal**) iff there is no y such that $x \rightarrow y$. We write $x \Downarrow^k y$ iff $x \stackrel{k}{\rightarrow} y$ and y is normal. If there is any k with $x \Downarrow^k y$, we simply write $x \Downarrow y$.

Corollary 2.7 Let $\rightarrow$ be an uniformly confluent relation. If $x \Downarrow^{k_1} y_1$ and $x \Downarrow^{k_2} y_2$, then $k_1 = k_2$ and $y_1 = y_2$.

Proof By uniform confluence, we know that there is a term z and numbers l_1 and l_2 such that $y_1 \stackrel{l_1}{\rightarrow} z \stackrel{l_2}{\leftarrow} y_2$ and $k_1 + l_1 = k_2 + l_2$. But since y_1 and y_2 are irreducible, they can not reduce. So $l_1 = l_2 = 0$ and $y_1 = z = y_2$ and therefore $k_1 = k_2$. ■

So in *uniformly* confluent systems, the normal form of a given term (if existing) and the *length* of a normalizing reduction chain are independent of the reduction strategy.

Chapter 3

L: Weak Call-by-Value λ-Calculus

L is a weak call-by-value λ-calculus. Those restrictions simplify the semantics compared to $\lambda\beta$ without restricting the computational capabilities.

It was used in [For14] as a formal model of computation to allow rigorous proofs in the context of computability theory.

We briefly restate the properties of L relevant for this thesis. We omit proofs that can be found in [For14].

Definition 3.1 (Term)

$$\mathrm{L} \ni s, t, u ::= x \mid \lambda s \mid st \qquad (x \in \mathbb{N})$$

We call x **variable**, λs is an **abstraction** with **body** s and st is the **application** of s to t. We omit parentheses according to $(st)u = stu$, $\lambda(st) = \lambda st$ and $s(\lambda t) = s\lambda t$.

We use **De Bruijn indices** for the binding of variables: Abstractions do not explicitly name the variable they bind, but a variable counts the number of other abstractions between the variable and the binding abstraction.

Example 3.2 The term $\lambda x.x\lambda y.xy$ (in named representation) is given by $\lambda 0 \lambda 10$ using De Bruijn indices.

We compare the differences of De Bruijn and named representation, with respect to the formalization, in Section 3.3. Since named terms are more human readable, we will sometimes use named terms for examples. They can be mechanically translated to De Bruijn indices.

The semantics of L is defined in terms of a substitution and a reduction.

Definition 3.3 (Substitution)

$$\begin{aligned} x^x_u &:= u \\ y^x_u &:= y \quad \text{if } y \neq x \\ (st)^x_u &:= s^x_u t^x_u \\ (\lambda s)^x_u &:= \lambda(s^{1+x}_u). \end{aligned}$$

Definition 3.4 (Reduction)

$$\frac{}{(\lambda s)(\lambda t) \succ_{\mathrm{L}} s^0_{\lambda t}}\ \beta \qquad \frac{s \succ_{\mathrm{L}} s'}{st \succ_{\mathrm{L}} s't}\ \textsc{AppL} \qquad \frac{s \succ_{\mathrm{L}} t'}{st \succ_{\mathrm{L}} st'}\ \textsc{AppR}$$

Thus, a **redex** in L is each term $(\lambda s)(\lambda t)$ that does not appear under an abstraction.

A term s is **closed** iff for all x and u we have $s^x_u = s$. This coincides with the intuition of s having no free variable. A **procedure** is a closed abstraction.

Some basic properties follow without even looking at the definition of substitution:

Fact 3.5 If $s \succ^k_{\mathrm{L}} s'$ and $t \succ^l_{\mathrm{L}} t'$, then $st \succ^{k+l}_{\mathrm{L}} s't'$.

Theorem 3.6 $\succ_{\mathrm{L}}$ is uniform confluent.

Proof By Theorem 2.4, the claim is equivalent to the uniform diamond property. The proof of this can be found in [For14] and is similar to the proofs of Theorem 4.7 and Lemma 6.6. ∎

Note that substitution in L is capturing: when substituting a term u with unbound variables into the body of an abstraction, we do not adjust the variables in the abstraction in any way. This simplification compared to $\lambda\beta$ is possible since L is *weak*, that is that we can not reduce in the body of abstraction. Combined with the fact that we are only interested in a reasonable semantics for terms closed on top-level, we never substitute variables with non-closed terms.

L is call-by-value as we require the argument in a β-reduction to be irreducible (see Fact 3.7).

Fact 3.7 A closed term is irreducible iff it is an abstraction.

To reason about non-closed terms, we refine the notion of closedness: A term is k-closed, if it only references variables less than k:

Definition 3.8 (k-closed)

$$\frac{x < k}{k\text{-closed } x} \qquad \frac{(1+k)\text{-closed } s}{k\text{-closed } (\lambda s)} \qquad \frac{k\text{-closed } s \quad k\text{-closed } t}{k\text{-closed } (st)}$$

We need the following properties in this thesis:

Fact 3.9 s is closed iff s is 0-closed.

Fact 3.10 If s is k-closed and $k' \geq k$, then s is k'-closed.

Fact 3.11 A closed term is k-closed.

Fact 3.12 If s is (k+1)-closed and t is k-closed, then s_t^k is k-closed.

We define the **size** of a term since we sometimes will use induction on the size:

$$
\begin{aligned}
|x| &:= 0 \\
|st| &:= 1 + |s| + |t| \\
|\lambda s| &:= 1 + |s|
\end{aligned}
$$

3.1 L as Programming Language

As a justification to use L as a functional Programming language, we briefly outline how to represent data and perform recursion in L. More details can be found in [For14].

Data in L can be represented via the **Scott encoding**: Assume an inductive data type T with constructors $c_1, \ldots, c_k$ each taking l_i arguments. Then the Scott encoding of $x = c_i a_1 \ldots a_{l_i}$ is the procedure $\overline{x} = \lambda c_1 \cdots c_k . c_i \,\overline{a_1} \ldots \overline{a_{l_i}}$. The constructor c_i itself can be represented by the procedure $\lambda a_1 \cdots a_{l_i} . \lambda c_1 \cdots c_k . c_i a_1 \cdots a_{l_i}$.

Example 3.13 The Scott encoding of a natural number k and the successor constructor succ are defined as:

$$
\begin{aligned}
\overline{0} &:= \lambda zs.z \\
\overline{k+1} &:= \lambda zs.s\overline{k} \\
\overline{\text{succ}} &:= \lambda n.\lambda zs.sn
\end{aligned}
$$

Thus data is represented by the procedure performing the match-operation on the data. In the same way, L-terms itself can be represented in L as data.

To use a recursive procedure, we first define the recursion combinator.

Definition 3.14 (Recursion Combinator)

$$
\begin{aligned}
r &:= \lambda rf.f(\lambda x.rrfx) \\
\rho s &:= \lambda x.rrsx
\end{aligned}
$$

Note that ρ is not an L-term, but a meta-function.

To write a recursive procedure, we can write a non-recursive procedure s taking an additional first argument that is used for recursive calls. Then ρs is the desired recursive procedure:

Fact 3.15 If s is a procedure, then ρs is a procedure.

Fact 3.16 If s and t are procedures, then $\rho s\, t \succ_{\mathrm{L}}^{3} s(\rho s)t$.

This way, we can define for example addition on natural numbers:

$$\overline{\mathrm{add}} := \rho(\lambda fmn.mn(\lambda m'.\overline{\mathrm{succ}}(fm'n)))$$

This definition directly corresponds to the definition of addition in Coq itself:

We take two arguments m and n and want to compute $m+n$. Therefore, we perform pattern-matching on m: In the first case, where $m = 0$, we return n. In the second case, we have $m = \mathrm{succ}\ m'$, use recursion and return succ $(m' + n)$.

The definition is formally justified by the correctness lemma:

Fact 3.17 $\overline{\mathrm{add}}$ is a procedure and $\overline{\mathrm{add}}\,\overline{m}\,\overline{n} \succ_{\mathrm{L}}^{*} \overline{m+n}$

Therefore, we say that addition is **internalized** by the L-term $\overline{\mathrm{add}}$. Note that also the constructor succ is internalized by $\overline{\mathrm{succ}}$

3.2 Semi-Automated Internalization

Together with Yannick Forster, we developed a framework that automates the internalization of functions: Yannick Forster mostly worked on Coq-tactics that transform a Coq-definition into an L-term, and we developed Coq-tactics that semi-automate the proof of the correctness lemma. The framework makes use of already internalized functions and their correctness lemmas when internalizing new functions.

To formally reason about a function in Coq, it first has to be defined. The semi-automated internalization eases the task to define nearly the same function again in L, to prove that it is L-computable.

More details can be found in [For15].

3.3 Named L

The version of L with explicit names just differs from the De Bruijn-version in some details:

$$s, t, u \ ::= \ x \ \mid \ \lambda x.s \ \mid \ st \qquad (x \in \mathbb{N})$$

$$
\begin{aligned}
x^x_u &:= u \\
y^x_u &:= y && \text{if } y \neq x \\
(st)^x_u &:= s^x_u t^x_u \\
(\lambda x.s)^x_u &:= \lambda x.s \\
(\lambda y.s)^x_u &:= \lambda y.s^x_u && \text{if } y \neq x
\end{aligned}
$$

$$
\frac{}{(\lambda x.s)(\lambda y.t) \succ_{\mathrm{L}} s^x_{\lambda y.t}}\ \beta \qquad \frac{s \succ_{\mathrm{L}} s'}{st \succ_{\mathrm{L}} s't}\ \textsc{AppL} \qquad \frac{s \succ_{\mathrm{L}} t'}{st \succ_{\mathrm{L}} st'}\ \textsc{AppR}
$$

The benefit of named L is that the representation is more human readable. For each abstraction, all uses of the bound variable are denoted by syntactical identical variables. That was not the case with De Bruijn indices (see Example 3.2). Note that the concrete set of variables in named L, in our case $\mathbb{N}$, does not matter as long it is infinite (and has decidable equality).

When we speak about named L in this thesis, we will always do so modulo α-conversion: Each syntactic term represents the equivalence class of all terms obtained by consistently renaming bound variables. So $\lambda x.x$ and $\lambda y.y$ represent the same object.

But this way of handling α-equivalence is not possible in Coq without notable overhead. Therefore we studied to what extend is is possible to formalize named L just using syntactic equality, without any notion of α-equivalence. In $\lambda\beta$, this is not possible since substitution may has to rename variables when substituting with non-closed terms.

In our setting, we were able to convert the whole Coq formalization of [For14] to named L. The only major change was introducing a notion of k-closedness that does not contain an index, but the set of all unbound variables.

All other proofs remained nearly the same. Interestingly, uniform confluence still holds in named L, even without a notion of α-equivalence.

One problem that we were faced with is that for Scott encoded datatypes, it now is important how the different variables are named. Since at this point, the Coq formalization in [For14] did not use the automation described in Section 3.2, we had to manually alter a lot of correctness proofs.

Overall, we decided to stick with using De Bruijn indices, since a formalization of α-equivalence would introduce too much overhead.

Note that on closed terms, the translation from the De Bruijn representation to th named one and vice versa can be easily defined. Therefore, the results on closed terms can be translated without any problems between the two variants of L.

Chapter 4

SKv: Call-by-Value Combinatory Logic

Combinatory Logic, although initially introduced to define first-order logic without the need for bound variables [Sch24], can be used to eliminate abstractions and substitutions in $\lambda\beta$.

Similar to L beeing a call-by-value variant of $\lambda\beta$, we introduce SKv, a call-by-value variant of the combinatory logic SKI. We will first show some basic properties of SKv. In Chapter 5, we the prove that SKv can be used to eliminate abstractions and substitutions from L.

Definition 4.1 (Term)

$$X, Y, Z \ ::= \ x \ \mid \ \mathbf{K} \ \mid \ \mathbf{S} \ \mid \ XY \qquad (x \in \mathbb{N})$$

The set of all x occurring in X is denoted by $\mathrm{FV}\,(X)$ and a term X is **closed** iff $\mathrm{FV}\,(X)$ is empty. We call **S** and **K** **combinators**. Their semantics will be similar to a function taking multiple arguments. Therefore, a **value** is a variable or a combinator not applied to enough other values:

$$X, Y \ ::= \ x \ \mid \ \mathbf{K} \ \mid \ \mathbf{K}X \ \mid \ \mathbf{S} \ \mid \ \mathbf{S}X \ \mid \ \mathbf{S}XY \qquad (x \in \mathbb{N})$$

The semantics of SKv is defined in terms of the reduction relation:

Definition 4.2 (Reduction)

$$\frac{X, Y \text{ values}}{\mathbf{K}XY \succ_{\mathrm{SK}} X}\ \mathrm{K} \qquad \frac{X, Y, Z \text{ values}}{\mathbf{S}XYZ \succ_{\mathrm{SK}} XZ(YZ)}\ \mathrm{S} \qquad \frac{X \succ_{\mathrm{SK}} X'}{XY \succ_{\mathrm{SK}} X'Y}\ \textsc{AppL}$$

$$\frac{Y \succ_{\mathrm{SK}} Y'}{XY \succ_{\mathrm{SK}} XY'}\ \textsc{AppR}$$

Note that for all redexes, that are terms of form $\mathbf{K}XY$ and $\mathbf{S}XYZ$ with values X,Y and Z, each proper subterm is a value.

We define the identity combinator $\mathbf{I} := \mathbf{SKK}$. Note that $\mathbf{I}$ has indeed the properties of the identity, as for all values X, we have $\mathbf{I}X \succ_{\text{SK}}^2 X$. In contrast to SKI in [HS08], we did not choose to include the identity as a basic combinator in the definition of SKv.

Fact 4.3 If $X \succ_{\text{SK}}^k X'$ and $Y \succ_{\text{SK}}^l Y'$, then $XY \succ_{\text{SK}}^{k+l} X'Y'$.

The notion of values is justified:

Fact 4.4 Every value is irreducible.

Note that the converse does not hold, since for example xK is irreducible and not a value. But since for our final results, we are mostly interested in closed terms, we define the interaction between values and variables such that convenient substitution lemmas hold, for example Fact 4.14.

For closed terms, we have the expected property:

Fact 4.5 A closed term is irreducible iff it is a value.

Corollary 4.6 If X is a value and $X \succ_{\text{SK}}^* Y$, then $X = Y$.

Confluence now follows by the same argument as for L in Theorem 3.6.

Theorem 4.7 $\succ_{\text{SK}}$ is uniformly confluent.

Proof By Theorem 2.4, uniform confluence is equivalent to the diamond property.

So assuming $X \succ_{\text{SK}} Y_1$ and $X \succ_{\text{SK}} Y_2$, we show that either $Y_1 = Y_2$ or there exists Z with $Y_1 \succ_{\text{SK}} Z$ and $Y_2 \succ_{\text{SK}} Z$.

All proper subterms of redexes are values and therefore, redexes are not nested. So either the two reductions $Y_1 \succ_{\text{SK}} Z$ and $Y_2 \succ_{\text{SK}} Z$ already deterministicly contracted the same redex and we have $Y_1 = Y_2$. Or they contracted non-overlapping redexes and we can join Y_1 and Y_2 by contracting the other redex as well. ∎

By Corollary 2.6, we conclude:

Corollary 4.8 $\succ_{\text{SK}}$ is Church-Rosser.

Together with Fact 4.4, this yields two corollaries:

Corollary 4.9 If Y is a value and $X \equiv_{\text{SK}} Y$, then $X \succ_{\text{SK}}^* Y$.

Corollary 4.10 If X and Y are values and $X \equiv_{\text{SK}} Y$, then $X = Y$.

4.1 Miscellaneous

We close this chapter with some more specific properties of SKv needed in later chapters.

Since SKv is call-by-value, each subterm of a redex must be normalized. So we can decompose reduction paths:

Lemma 4.11 For every reduction $X \succ_{\mathrm{SK}}^{k} Y$ with $k > 0$ there are $X_1X_2 = X$ and Y_1, Y_2 with $X_i \succ_{\mathrm{SK}}^{k_i} Y_i$ such that one of two cases holds:

Either $Y = Y_1Y_2$ with $k = k_1 + k_2$, or the Y_i are values and there are Z and k' with $Y_1Y_2 \succ_{\mathrm{SK}} Z$ and $Z \succ_{\mathrm{SK}}^{k'} Y$, where $k = k_1 + k_2 + 1 + k'$.

Proof If X reduces, it is an application $X = X_1X_2$. Then either somewhere in the reduction path there is the first redex on top level and the second case suffices. Or there is no redex at all on toplevel and the first case suffices. ∎

Fact 4.12 If X is closed and $X \succ_{\mathrm{SK}}^{*} X'$, then X' is closed.

We define **substitution** in a straightforward manner:

$$\begin{aligned} x^x_Y &:= Y \\ (XY)^x_Z &:= X^x_Z Y^x_Z \\ X^x_Y &:= X \qquad \text{otherwise} \end{aligned}$$

Definition 4.13 X is a **maximal value** if X is a value, but XY is not for any Y.

So maximal values are variables, $\mathbf{S}XY$ or $\mathbf{K}X$. The motivation is that we can substitute variables for other maximal values without affecting valueness:

Fact 4.14 If Y is a maximal value, then X is a value iff X^x_Y is a value.

For non-maximal values, this does not hold since for example $x\mathbf{K}$ is not a value, but $(x\mathbf{K})^x_{\mathbf{K}} = \mathbf{KK}$ is a value.

Fact 4.15 If X and Y are values, the successor of XY is unique. That is, from $(XY) \succ_{\mathrm{SK}} Z_1$ and $(XY) \succ_{\mathrm{SK}} Z_2$, we know $Z_1 = Z_2$.

Fact 4.16 If Y is closed and $x \neq z$, then $z \in \mathrm{FV}(X)$ iff $z \in \mathrm{FV}(X^x_Y)$.

We define the **size** of a term since we sometimes will use induction on the size, not just the structure of a term:

$$\begin{aligned} |XY| &:= 1 + |X| + |Y| \\ |X| &:= 0 \qquad \text{otherwise} \end{aligned}$$

Chapter 5

Embedding L into SKv

It is known that $\lambda\beta$ and SKI are strongly related [HS08]. But there is no known mapping from $\lambda\beta$ to SKI that is fully compatible with the intensional notions of equivalence, not even for the broader notion of equivalence in $\lambda\beta\eta$

In a call-by-value setting, this is different: We give a compiler $\mathcal{C}$ from L to SKv that preserves irreducibility and is fully compatible with our non-extensional term equivalences, that is $s \equiv_{\mathrm{L}} t$ iff $\mathcal{C}\, s \equiv_{\mathrm{SK}} \mathcal{C}\, t$. So L can be fully embedded into SKv.

Several undecidable, extensional properties in L like term equivalence or evaluation are known, so this shows that the corresponding problem in SKv is undecidable as well.

In this chapter, we will assume a named variant of L (see Section 3.3) instead of De Bruijn indices, so the syntactic representation of an L-term in this chapter is only unique up to consistent renaming of bound variables.

There is not much insight in using De Brujin indices here, but handling the indices distracts from the main idea of the proofs. The Coq formalization of this chapter still uses De Brujin indices, so the differing details can be found there.

5.1 Pseudo-Abstraction

We will give a computable function $[x].X$ called pseudo-abstraction of x with respect to X. Pseudo-abstractions have properties similar to abstractions in L.

Definition 5.1 (Pseudo-Abstraction)

$$\begin{aligned} [x].x &:= \mathbf{I} & \\ [x].X &:= \mathbf{K}X && \text{if } x \notin \mathrm{FV}\,(X) \text{ and } X \text{ value} \\ [x].(XY) &:= \mathbf{S}([x].X)([x].Y) && \text{otherwise} \end{aligned}$$

Fact 5.2 $[x].X$ is a maximal value.

This relates to the fact that abstractions in L are irreducible, but may reduce if applied to another value.

The pseudo-abstractions presented in [HS08] do not have this property. The difference to our pseudo-abstraction is that the third equation of Definition 5.1 applies to all non-values. This 'lifting' with **S** turns non-values into values: While in [HS08], the pseudo-abstraction of **II** is **K**(**II**) and not a value, we have $[x].\mathbf{II} = \mathbf{SII}$, which is a value.

The notion of 'pseudo-abstraction' is justified and a pseudo-abstraction similar to an L-abstraction:

Theorem 5.3 (Correctness) For all values Y, we have $([x].X)Y \succ^{+}_{\mathrm{SK}} X^{x}_{Y}$

Proof Induction on X. In all cases that lead to the first two equations in Definition 5.1, the claim follows by the semantics of SKv. Otherwise, we know that $X = X_1X_2$ and the third equation in Definition 5.1 applies. By the induction hypothesis, $([x].X_i)Y \succ^{*}_{\mathrm{SK}} {X_i}^{x}_{Y}$ for $i = 1, 2$, which together with Fact 5.2 concludes the claim:

$$\begin{aligned} ([x].X_1X_2)Y &= \mathbf{S}([x].X_1)([x].X_1)Y \\ &\succ_{\mathrm{SK}} ([x].X_1)Y(([x].X_1)Y) \\ &\succ^{*}_{\mathrm{SK}} {X_1}^{x}_{Y}{X_2}^{x}_{Y} \\ &= (X_1X_2)^{x}_{Y} \end{aligned}$$

■

On certain terms, substitution and pseudo abstraction commute:

Lemma 5.4 Let Y be a closed, maximal value and $x \neq z$. Then $([z].X)^{x}_{Y} = [z].(X^{x}_{Y})$.

Proof Induction on X. For variables $X = z \neq x$, the equation is trivial. Otherwise, by Fact 4.14 and Fact 4.16, we know that the conditions that $z \notin \mathrm{FV}(X)$ and X is a value hold if and only if the conditions $z \notin \mathrm{FV}(X^{x}_{Y})$ and X^{x}_{Y} is a value. Therefore the same equations in the definition of the pseudo-abstraction are applicable to both sides:

- Assuming $z \notin \mathrm{FV}(X)$ and X is a value, we get $([z].X)^{x}_{Y} = (\mathbf{K}X)^{x}_{Y} = \mathbf{K}X^{x}_{Y} = [z].(X^{x}_{Y})$
- Otherwise, $X = X_1X_2$ with the induction hypothesis $([z].X_i)^{x}_{Y} = [z].\,{X_i}^{x}_{Y}$ for $i = 1, 2$. We get $([z].X_1X_2)^{x}_{Y} = (\mathbf{S}([z].X_1)([z].X_2))^{x}_{Y} = \mathbf{S}([z].X_1)^{x}_{Y}([z].X_2)^{x}_{Y} = [z].((X_1X_2)^{x}_{Y})$ ■

Note that the side conditions in the definition of pseudo-abstraction are chosen carefully. For other variants resembling the ones from [HS08], Lemma 5.4 does not hold since it may be the case that different cases of Definition 5.1 apply to X and X^{x}_{Y}. Also the precondition that Y is a maximal value in Lemma 5.4 is required, beeing a maximal value is required as otherwise, Fact 4.14 is not applicable.

5.2 Compiling L into SKv

We can now define the SKv-term corresponding to a given L-term:

$$\begin{aligned} \mathcal{C}\,x &:= x \\ \mathcal{C}\,(st) &:= (\mathcal{C}\,s)\ (\mathcal{C}\,t) \\ \mathcal{C}\,(\lambda x.s) &:= [x].(\mathcal{C}\,s) \end{aligned}$$

In proofs, we write $\underline{s}$ instead of $\mathcal{C}\,s$ for readability. This also allows to drop parenthesis.

Fact 5.5 s is closed iff $\mathcal{C}\,s$ is closed.

Note that on the left, we speak about L-closedness, while on the right, we have SKv-closedness.

Fact 5.6 A closed term s is an abstraction iff $\mathcal{C}\,s$ is a value.

The closedness of s is required since for example $\mathcal{C}\,x$ is a value, but x is not an abstraction.

Theorem 5.7 A closed term s is irreducible iff $\mathcal{C}\,s$ is irreducible.

Proof Fact 5.6 with Fact 4.4 and Fact 3.7. ■

Fact 5.8 $(\mathcal{C}\,s)\ Y$ is not a value.

For the soundness of $\mathcal{C}$, the following compatibility lemma with SKv- and L-substitution is essential:

Lemma 5.9 If t is a procedure, then $(\mathcal{C}\,s)^x_{\mathcal{C}\,t} = \mathcal{C}\,(s^x_t)$.

Proof Induction on s. The cases for variables and applications are trivial. In the remaining case $s = \lambda z.s'$, we have the induction hypothesis $\underline{s'}^x_{\underline{t}} = \underline{(s'^x_t)}$.

We want to prove $([z].\underline{s'})^x_{\underline{t}} = \underline{(\lambda z.s')^x_t}$ with $x \neq z$. This follows by Lemma 5.4 with Fact 5.2 and the induction hypothesis. ■

Proposition 5.10 (Soundness) For all closed terms s with $s \succ_{\text{L}} t$, we have $\mathcal{C}\,s \succ^+_{\text{SK}} \mathcal{C}\,t$.

Proof Induction on $s \succ_{\text{L}} t$. All cases except the base case are straightforward.

In the base case, we have $s = (\lambda x.s')u$ where u is a procedure, so $t = s'^x_u$. We need to prove $\underline{(\lambda x.s')u} \succ^+_{\text{SK}} \underline{s'^x_u}$. This holds by using Lemma 5.9 and Theorem 5.3: $\underline{(\lambda x.s')u} = ([x].\underline{s'})\underline{u} \succ^+_{\text{SK}} \underline{s'}^x_{\underline{u}} = \underline{(s'^x_u)}$ ■

Corollary 5.11 If s is closed and $s \succ^*_{\text{L}} t$, then $\mathcal{C}\,s \succ^*_{\text{SK}} \mathcal{C}\,t$.

Corollary 5.12 If s and t are closed and $s \equiv_{\text{L}} t$, then $\mathcal{C}\,s \equiv_{\text{SK}} \mathcal{C}\,t$.

Together with Theorem 5.7, we also conclude:

Fact 5.13 If s is closed and $s \Downarrow t$, then $\mathcal{C}\,s \Downarrow \mathcal{C}\,t$.

5.3 $\mathcal{C}$ is invertible

By Fact 5.13, we may normalize $\mathcal{C}\,s$ to get $\mathcal{C}\,t$, where t is the normal form of s. But for this property to be even more useful, we need a left-inverse of $\mathcal{C}$ to pull back the term t from $\mathcal{C}\,t$.

We first give the left-inverse of our pseudo-abstraction:

Definition 5.14

$$\begin{aligned}
[x]^{-1}.(\mathbf{SKK}) &:= x \\
[x]^{-1}.(\mathbf{S}XY) &:= ([x]^{-1}.X)([x]^{-1}.Y) \\
[x]^{-1}.(\mathbf{K}X) &:= X
\end{aligned}$$

Note that this is only a partial function, but for our results, the result on other arguments does not matter.

Proposition 5.15 $[x]^{-1}.([x].X) = X$

Proof Induction on X. ∎

Therefore, the pseudo-abstraction is injective.

Definition 5.16 (Left Inverse of $\mathcal{C}$)

$$\begin{aligned}
\mathrm{C}^{\text{-}1}\,x &:= x \\
\mathrm{C}^{\text{-}1}\,X &:= \lambda x.(\mathrm{C}^{\text{-}1}\,([x]^{-1}.X)) && \text{if } X \text{ is a value} \\
\mathrm{C}^{\text{-}1}\,(XY) &:= (\mathrm{C}^{\text{-}1}\,X)(\mathrm{C}^{\text{-}1}\,Y)
\end{aligned}$$

In the second rule, x is any fresh variable. As we identify named L-terms modulo α-equivalence, this still defines a function.

The definition is well-founded since the inverse of pseudo-abstraction decreases the size of arguments that are no variables.

Proposition 5.17 $\mathrm{C}^{\text{-}1}\,(\mathcal{C}\,s) = s$

Proof Induction on the size of s, using Fact 5.6 and Fact 5.8 to show that the corresponding equations in the definitions of $\mathcal{C}$ and $\mathrm{C}^{\text{-}1}$ apply. ∎

We have now shown that $\mathcal{C}$ is injective and that we can compute the normal form of s, if it exists, by normalizing $\mathcal{C}\,s$ to X and the computing $\mathrm{C}^{\text{-}1}\,X$. But we have not proven that for a diverging term s, the compilation $\mathcal{C}\,s$ diverges as well.

5.4 $\mathcal{C}$ is Complete

We now want to prove that the compilation is complete, that is that $\underline{s} \succ_{\text{SK}} \underline{t}$ translates back to $s \equiv_{\text{L}} t$.

Note that a stronger notion of completeness like $\underline{s} \succ_{\text{SK}} \underline{t} \Rightarrow s \succ_{\text{L}}^{*} t$ (similar to Proposition 5.10) does not hold:

Example 5.18 For a procedure u, we have $\underline{(\lambda x.xx)u} = \mathbf{SII}\underline{u} \succ_{\text{SK}} (\mathbf{I}\underline{u})(\mathbf{I}\underline{u}) = \underline{((\lambda x.x)u)((\lambda x.x)u)}$, but we can not reduce $(\lambda x.xx)u$ to $((\lambda x.x)u)((\lambda x.x)u)$

Instead, a SKv-reduction of a term $\mathcal{C}\,s$ is just the prefix of a path corresponding to an L-reduction:

Lemma 5.19 For a closed term s with $\mathcal{C}\,s \succ_{\text{SK}} X$, there is t such that $s \succ_{\text{L}} t$ and $X \succ_{\text{SK}}^{*} \mathcal{C}\,t$.

Proof Induction on s. As it is closed and reducible, it must be an application $s = s_1 s_2$. We now distinguish how $\underline{s_1 s_2} \succ_{\text{SK}} X$ is derived:

- The derivation is due to AppL. So $X = X_1 \underline{s_2}$ with $\underline{s_1} \succ_{\text{SK}} X_1$, our induction hypothesis yields t' with $X_1 = \underline{t'}$ and $s_1 \succ_{\text{L}} t'$, so the claim follows with $t = t' s_2$.
- The case for AppR is symmetric.
- The cases for S and K can be handled uniformly: By Fact 5.6, we know that $s_1 = \lambda x.s_1'$ and $s_2 = \lambda y.s_2'$ are both procedures. So we have $\underline{s_1 s_2} = ([x].\underline{s_1'})\underline{\lambda y.s_2'} \succ_{\text{SK}} X$. Because of $\lambda x.s_1' s_2 \succ_{\text{L}} (s_1')^x_{s_2}$ with Proposition 5.10 we also know $\underline{s_1 s_2} \succ_{\text{SK}}^{+} \underline{s'^{\,x}_{1\,s_2}}$. By Fact 4.15, X is the same as the first sucessor in the reduction $\underline{s_1 s_2} \succ_{\text{SK}}^{+} \underline{s'^{\,x}_{1\,s_2}}$ and therefore X itself reduces to $\underline{s'^{\,x}_{1\,s_2}}$ as well.

  So the claim follows with $t = s'^{\,x}_{1\,s_2}$ ∎

This translates to arbitrary reduction paths:

Proposition 5.20 For a closed term s with $\mathcal{C}\,s \succ_{\text{SK}}^{*} X$, there is t such that $s \succ_{\text{L}}^{*} t$ and $X \succ_{\text{SK}}^{*} \mathcal{C}\,t$.

Proof Complete induction on the k in $\underline{s} \succ_{\text{SK}}^{k} X$. For k=0, we take $t = s$.

Otherwise, there is Y with $\underline{s} \succ_{\text{SK}} Y$ and $Y \succ_{\text{SK}}^{k} X$. By Proposition 5.20, there is k' and u such that $s \succ_{\text{L}} u$ and $Y \succ_{\text{SK}}^{k'} \underline{u}$. By the uniform confluence of SKv, we can join this reduction path with $Y \succ_{\text{SK}}^{k} X$ and get $l \leq k$ and X' such that $\underline{u} \succ_{\text{SK}}^{l} X'$ and $X \succ_{\text{SK}}^{*} X'$.

We now apply our induction hypothesis on $\underline{u} \succ_{\text{SK}}^{l} X'$. This yields t with $u \succ_{\text{L}}^{*} t$ and $X' \succ_{\text{SK}}^{*} \underline{t}$. With t, the claim holds: $s \succ_{\text{L}} u \succ_{\text{L}}^{*} t$ and $X \succ_{\text{SK}}^{*} X' \succ_{\text{SK}}^{*} \underline{t}$. ∎

Note that without the uniform confluence of SKv, the induction in the proof of Proposition 5.20 would not necessarily be well-founded.

Corollary 5.21 If s is closed and $\mathcal{C}\, s \Downarrow X$, then $s \Downarrow \mathrm{C}^{-1}\, X$.

Proof From Proposition 5.20 using Fact 5.6 we have t such that $\mathcal{C}\, t = X$ and $s \Downarrow t$. The claim follows the correctness of C^{-1}. ∎

From Corollary 5.21, we conclude completeness w.r.t closed, normalizing terms:

Corollary 5.22 If s is a closed term and $\mathcal{C}\, s \Downarrow \mathcal{C}\, t$, then $s \Downarrow t$.

Combined with Fact 5.13, this suffices to reduce the halting problem or term evaluation from L to SKv.

We want to go further and show that term equivalence, even for non-terminating terms, is preserved. Therefore, we show completeness for arbitrary, closed terms. To avoid the problems from Example 5.18, we first study the structure of reductions of the image of β-redexes under $\mathcal{C}$.

Lemma 5.23 Assume $\underline{(\lambda x.s)t} \succ^*_{\mathrm{SK}} \underline{u}$, where s is an abstraction, $\lambda x.s$ is closed and t is a procedure. Then $(\lambda x.s)t \equiv_{\mathrm{L}} u$

Proof Note that u is closed by Fact 5.5 because $([x].s)t$ is closed.

We know that $\underline{(\lambda x.s)t}$ reduces to $\underline{s}^x_{\underline{t}} = \underline{s^x_t}$ by Theorem 5.3 and Lemma 5.9. This even is the normal form by Fact 5.6, as s^x_t must be an abstraction.

Because of confluence, $\underline{u}$ has normal form $\underline{s^x_t}$ as well. By Corollary 5.22, we conclude $s^x_t \equiv_{\mathrm{L}} u$. Thus the claim holds by β-reducing $(\lambda x.s)t$. ∎

If s is not an abstraction, we have:

Lemma 5.24 Assume $\underline{(\lambda x.s)t} \succ_{\mathrm{SK}} Y$, where s is not an abstraction, $\lambda x.s$ is closed and t is a procedure. Then there exists a closed s' with $Y = \underline{s'}$ such that $(\lambda x.s)t \equiv_{\mathrm{L}} s'$.

Proof Because $\lambda x.s$ is closed, we distinguish three cases for $\underline{\lambda x.s} = [x].s$

- If s is a variable, then it must be the bound x. The assumed reduction of $\underline{(\lambda x.x)t} = \mathbf{I}\underline{t} = \mathbf{SKK}\underline{t}$ is deterministic since $\underline{t}$ is a value. So $Y = (\mathbf{K}\underline{t})(\mathbf{K}\underline{t})$ and the claim follows with $s' = (\lambda x.t)(\lambda x.t)$.

- If s is a closed, we have $\underline{(\lambda x.s)t} = \mathbf{K}\underline{s}\,\underline{t}$. Since $\underline{s}$ and $\underline{t}$ are values, the reduction $\mathbf{K}\underline{s}\,\underline{t} \succ_{\mathrm{SK}} Y$ is deterministic with $Y = \underline{s}$. Thus the claim follows with $s' = s$.

- If s is not closed and not a variable, $s = s_1 s_2$ is an application and x (and only x) occurs unbound in s. We have $\underline{(\lambda x.s_1 s_2)t} = \mathbf{S}([x].\,\underline{s_1})([x].\,\underline{s_2})\underline{t}$. Since $([x].\,\underline{s_1})$ and $\underline{t}$ are values, the successor Y of $\mathbf{S}([x].\underline{s_1})([x].\underline{s_2})\underline{t}$ is unique with $Y = (([x].\underline{s_1})\underline{t})(([x].\underline{s_2})\underline{t})$. Thus the claim follows for $s' = ((\lambda x.s_1)t)((\lambda x.s_2)t)$. ∎

Proposition 5.25 (Completeness) If s is closed and $\mathcal{C}\,s \succ_{\mathrm{SK}}^{*} \mathcal{C}\,t$, then $s \equiv_{\mathrm{L}} t$.

Proof Assume a closed term s such that $\underline{s} \succ_{\mathrm{SK}}^{l} \underline{t}$. We now perform induction on the lexicographical order on (l, s), so we may assume our induction hypothesis for any s', t' and l' with $\underline{s'} \succ_{\mathrm{SK}}^{l'} \underline{t'}$ as long as either $l' < l$, or $l' = l$ and $|s'| < |s|$. Note that t is closed since $s \succ_{\mathrm{L}}^{*} t$.

If s is an abstraction, then by Fact 5.6 we know that $\underline{s}$ is irreducible and thus $\underline{s} = \underline{t}$. By injectivity of $\mathcal{C}$ we have $s = t$ and the claim holds.

Otherwise, since s is closed, $s = s_1 s_2$ is an application. If t is an abstraction, the claim holds by Corollary 5.22. Otherwise, since t is closed, $s = t_1 t_2$ is an application.

So we have $\underline{s_1 s_2} \succ_{\mathrm{SK}}^{k} \underline{t_1 t_2}$ and distinguish cases with Lemma 4.11:

- For $k = 0$ we have $\underline{s_1 s_2} = \underline{t_1 t_2}$, and the claim follows by injectivity of $\mathcal{C}$.

- Assume $\underline{s_1} \succ_{\mathrm{SK}}^{k_1} \underline{t_1}$ and $\underline{s_2} \succ_{\mathrm{SK}}^{k_1} \underline{t_1}$ with $k = k_1 + k_2$. Since $|s_i| < |s|$, our induction hypothesis is applicable and yields $s_1 \equiv_{\mathrm{L}} t_1$ and $s_1 \equiv_{\mathrm{L}} t_1$. Thus the claim $s_1 s_2 \equiv_{\mathrm{L}} t_1 t_2$ holds.

- Assume $\underline{s_1} \succ_{\mathrm{SK}}^{k_1} X_1$, $\underline{s_2} \succ_{\mathrm{SK}}^{k_1} X_2$, $X_1 X_2 \succ_{\mathrm{SK}} Y$ and $Y \succ_{\mathrm{SK}}^{k'} \underline{t_1 t_2}$ with $k = k_1 + k_2 + 1 + k'$, where X_1 and X_2 are values.

  As for $i = 1, 2$ the term $\underline{s_i}$ converges to X_i, we can apply Corollary 5.21 and get procedures $\lambda x.u_i$ with $s_i \succ_{\mathrm{L}} \lambda x.u_i$ and $X_i = \underline{\lambda x.u_i}$. We know that $X_1 X_2 = \underline{\lambda x.u_1}\ \underline{\lambda x.u_2} \succ_{\mathrm{SK}} Y$ and distinguish whether u_1 is an abstraction:

  - If u_1 is an abstraction, the claim follows form $\underline{\lambda x.u_1}\ \underline{\lambda x.u_2} \succ_{\mathrm{SK}}^{*} \underline{t_1 t_2}$ with Lemma 5.23.
  - Otherwise, Lemma 5.24 on $\underline{\lambda x.u_1}\ \underline{\lambda x.u_2} \succ_{\mathrm{SK}} Y$ yields a closed s' with $Y = \underline{s'}$ and $(\lambda x.u_1)(\lambda x.u_2) \equiv_{\mathrm{L}} s'$.

    We now use our induction hypothesis on $Y = \underline{s'} \succ_{\mathrm{SK}}^{k'} \underline{t_1 t_2}$ which yields $s' \equiv_{\mathrm{L}} t_1 t_2$. Thus the claim holds. ■

5.5 Conclusion

So the structure of L is fully captured in SKv using $\mathcal{C}$:

Theorem 5.26 If s and t are closed terms, then $s \equiv_{\mathrm{L}} t$ iff $\mathcal{C}\,s \equiv_{\mathrm{SK}} \mathcal{C}\,t$.

Proof One direction is Proposition 5.10.

For the reverse direction, we first use the Church-Rosser property on $\underline{s} \equiv_{\mathrm{SK}} \underline{t}$ to get Y with $\underline{s} \succ_{\mathrm{SK}}^{*} Y$ and $\underline{t} \succ_{\mathrm{SK}}^{*} Y$. From Proposition 5.20, we get u such that $Y \succ_{\mathrm{SK}}^{*} \underline{u}$. The claim follows via Proposition 5.25 used on $\underline{s} \succ_{\mathrm{SK}}^{*} \underline{u}$ and $\underline{t} \succ_{\mathrm{SK}}^{*} \underline{u}$. ■

Recall that Theorem 5.7 states that $\mathcal{C}$ preserves reducibility. So combined, we can conclude:

Theorem 5.27 If s is a closed term, then $s \Downarrow t$ iff $\mathcal{C}\, s \Downarrow \mathcal{C}\, t$

Thus we can say that we fully embedded L into SKv. That also means that each semantic property on L-terms can be reduced to a similar, semantic property on SKv-terms.

Chapter 6

LC: L with Closures

The substitution semantics for L in Chapter 3 is short and suitable for theoretical reasoning. But for reasons we point out, this semantics does not suggest an efficient interpreter for L.

To reason about the correctness of a more realistic interpreter, we introduce LC, a variant of L with closures. It was inspired both by explicit substitution semantics for λ-calculus [ACCL91] and variant of λ-calculus in [LM99]. We prove that LC is sound and complete with respect to L. We then argue why LC is computationally more efficient and give an interpreter for which we prove soundness and completeness. We prove that this interpreter can be used to interpret L as well.

6.1 Drawbacks of Substitution Semantics in L

One practical problem with a naive implementation of an interpreter performing substitutions is the need to traverse the whole body of the abstraction for each β-reduction.

Example 6.1 Let b_n be defined by $b_0 := \mathbf{I} := \lambda x.x$ and $b_{1+n} := \lambda x.b_n x$. Then $b_n\ \mathbf{I} \succ_{\mathrm{L}}^{n+1} \mathbf{I}$. But although $|b_n|$ is in $\theta(n)$, the naive interpreter needs to substitute into each of the b_i for $i = 0, \ldots, n$ taking time $|b_i|$ each, so the overall time required is in $O(n^2)$, where n is linear to the length of the reduction.

Also, the naive implementation using substitution even traverses closed subterms, for example Scott-encoded datatypes, and traverses subterms not effecting the final normal form. And every subterm under multiple abstractions is possibly traversed multiple times.

Besides those practical problems, the worst time complexity of an interpreter that fully prints the resulting term *must* be exponential as there are terms that grow exponentially in the number of performed reductions [LM08]:

Example 6.2 For arbitrary terms s and t, let $s^0 t := t$ and $s^{k+1} t := s(s^k t)$ and let $\underline{n} := \lambda x.\lambda y.x^n y$ be the Church numeral for n. Then $\underline{n}\,\underline{2}$ normalizes in linearly many steps, but has an exponentially large normal form.

The semantics of LC corresponds to a way functional programming languages in practice can be implemented without substitution, but using environments to remember bound variables, like for example in the formal specification of SML [MTHM97].

This approach only traverses those sub-expressions really needed for the evaluation, and only does so once. More details on the computational aspects can be found in Section 6.5.

6.2 Basic Properties of LC

We now define LC and give basic properties that do not even require reasoning using L.

Definition 6.3 (Term)

$$p, q, r \;::=\; x \;\mid\; s[\sigma] \;\mid\; p{\cdot}q \qquad (x \in \mathbb{N}, s \in \mathrm{L}, \sigma \in \text{list LC})$$

We write application in LC with $\cdot$ in between and omit parenthesis according to $(st)[\sigma] = st[\sigma]$, $p{\cdot}(s[\sigma]) = p{\cdot}s[\sigma]$, and $(\lambda s)[\sigma] = \lambda s[\sigma]$.

The term $s[\sigma]$ will be called a **closure**, and will refer to the subterm s as **body** and to the list σ as **environment** of the closure. We write $s[\,]$ for the term s in the empty environment.

A closure with an abstraction as body is a **value**. Values will correspond to abstractions in L.

Before we introduce the semantics of LC, we briefly specify the notations and operations on lists we require. We write $[\,]$ for the empty list and $a{::}A$ for the list obtained by prepending a in front of A. The length of A is denoted by $|\sigma|$. The nth element of a list (starting from 0) with default value d is defined as:

$$\begin{aligned} 0\text{-th}_d\,(a{::}A) &:= a \\ k\text{-th}_d\,[\,] &:= d \\ (k+1)\text{-th}_d\,(a{::}A) &:= k\text{-th}_d\,A \end{aligned}$$

Since we use De Bruijn indices, the list σ represents a substitution: The term corresponding to x in the environment σ is $x\text{-th}_x\,\sigma$.

We can now define the semantics of LC:

Definition 6.4 (Reduction)

$$\frac{}{x[\sigma] \succ_{\mathrm{LC}} x\text{-th}_x\,\sigma}\ \textsc{Var} \qquad \frac{}{\lambda s[\sigma]\cdot\lambda t[\tau] \succ_{\mathrm{LC}} s[\lambda t[\tau]::\sigma]}\ \beta$$

$$\frac{}{st[\sigma] \succ_{\mathrm{LC}} s[\sigma]\cdot t[\sigma]}\ \textsc{App} \qquad \frac{p \succ_{\mathrm{LC}} p'}{p\cdot q \succ_{\mathrm{LC}} p'\cdot q}\ \textsc{AppL} \qquad \frac{q \succ_{\mathrm{LC}} q'}{p\cdot q \succ_{\mathrm{LC}} p\cdot q'}\ \textsc{AppR}$$

The idea is that β-reduction in LC corresponds to β-reduction in L. The reductions App and Var are **simplification steps**, as they correspond to performing the L-substitution in small, parallel steps. We will define a function performing multiple simplification steps later. We did not integrate this function into β-reduction since our more fine-grained semantics eases inductive reasoning.

In contrast to L, where substitution traverses the whole body of an abstraction, a value in LC is just traversed as deep as currently needed: We must not push the environment under an abstraction, but only as far as needed to verify whether a term is a value.

We conclude this section with basic properties independent of L:

Fact 6.5 If $p \succ^k_{\mathrm{LC}} p'$ and $q \succ^l_{\mathrm{LC}} q'$ then $p\cdot q \succ^{k+l}_{\mathrm{LC}} p'\cdot q'$.

Lemma 6.6 $\succ_{\mathrm{LC}}$ is uniform confluent.

Proof By Theorem 2.4, uniform confluence is equivalent to the uniform diamond property.

So under the assumption that $p \succ_{\mathrm{LC}} q_1$ and $p \succ_{\mathrm{LC}} q_2$, we show that either $q_1 = q_2$ or there exists r with $q_1 \succ_{\mathrm{LC}} r$ and $q_2 \succ_{\mathrm{LC}} r$.

As the redexes in the definition of $\succ_{\mathrm{LC}}$ do not overlap and are deterministic. Therefore either the two reductions $p \succ_{\mathrm{LC}} q_1$ and $p \succ_{\mathrm{LC}} q_2$ either contracted the same redex and $q_1 = q_2$. Or they contracted non-overlapping redexes and we can get r by reducing both redexes. ∎

With Corollary 2.6 we conclude:

Corollary 6.7 $\succ_{\mathrm{LC}}$ is Church-Rosser.

6.3 Connecting LC and L

Our intention is that an LC-reduction path of $s[\,]$ correspond to an L-reduction paths of s itself and vice versa. To formalize this, we introduce the L-term corresponding to a given LC-term.

We first of all notice that, by the semantics of LC, only a subset of syntactically possible terms can possibly be reduced from a closed L-term in the empty environment:

Definition 6.8 (Admissible)

$$\frac{|\sigma|\text{-closed } s \quad \forall p \in \sigma, \text{admissible } p \wedge \text{value } p}{\text{admissible } s[\sigma]} \qquad \frac{\text{admissible } p \quad \text{admissible } q}{\text{admissible } p \cdot q}$$

Note that LC-variables are never admissible and any closed L-term in the empty environment is admissible.

Fact 6.9 If $p \succ_{\text{LC}}^* q$ and p is admissible, then q is admissible.

We can now state in which sense our definition of a value is justified:

Fact 6.10 An admissible LC-term is a value iff it is irreducible.

We now want to translate LC terms into corresponding L-terms. For a closure, we will first translate the environment and substitute the resulted list of L-terms into the body. As we use De Bruijn indices, this can be performed elegantly by the **parallel substitution** $(\cdot)\{\cdot\}^{\cdot} : \text{L} \to \text{list L} \to \mathbb{N} \to \text{L}$, defined by:

$$\begin{aligned} x\{A\}^k &:= (x-k)\text{-th}_x\, A && \text{if } k \le x \\ x\{A\}^k &:= x && \text{otherwise} \\ (st)\{A\}^k &:= s\{A\}^k\; t\{A\}^k \\ (\lambda s)\{A\}^k &:= \lambda(s\{A\}^{k+1}) \end{aligned}$$

The parameter k corresponds to the fact that when substituting under k abstractions, we do not want to substitute De Bruijn indices less than k.

Parallel substitution satisfies two characteristic equations:

Fact 6.11 $s\{[\,]\}^x = s$

Lemma 6.12 $s\{t{::}A\}^k = (s\{A\}^{k+1})_t^k$ for any list A of closed L-terms.

Proof By induction on s. The cases for application and abstraction follow directly from the induction hypothesis. In the last case $s = x$ is a variable. By case distinction, the claimed equality $x\{t{::}A\}^k = (x\{A\}^{k+1})_t^k$ holds:

- For $k > x$, both sides are x.
- For $k = x$, both sides are t.
- For $k < x$ and $|t{::}A| < x - k$, both sides are x.
- Otherwise, we have $k < x \le |t{::}A| + k$ and therefore, both parallel substitutions yield the same element of A. The claim holds since all elements in A are closed, so the substitution of k for t on the right does not change the term. ∎

As expected, the result of a parallel substitution is closed if the list is long enough and closed:

Lemma 6.13 Let A be a list of closed L-terms. If $s \in \mathrm{L}$ is a $|A|$-closed, then $s\{A\}^0$ is closed.

Proof We generalize the statement: If s is $(y + |A|)$-closed, then $s\{A\}^y$ is y-closed. From that, the claim follows with $k = 0$ using Fact 3.9.

The generalized claim holds by induction on A using the two characteristic equations for the parallel substitution, Fact 6.11 and Lemma 6.12, as well as Fact 3.12 and Fact 3.11. ∎

We now can finally define the **translation** $\lceil\cdot\rceil : \mathrm{LC} \to \mathrm{L}$, the L-term corresponding to an LC-term:

$$\begin{aligned} \lceil x \rceil &:= x \\ \lceil p{\cdot}q \rceil &:= \lceil p \rceil \lceil q \rceil \\ \lceil s[\sigma] \rceil &:= s\{\lceil \sigma \rceil\}^0, \end{aligned}$$

where $\lceil \sigma \rceil$ is the pointwise translation of σ.

Proposition 6.14 If p is admissible, then $\lceil p \rceil$ is closed.

Proof By induction on p. Since p is admissible, is must be an application or a closure. The first case is trivial. In the second case, we know that $p = s[\sigma]$ is admissible, therefore s is $|\sigma|$-closed. By the induction hypothesis, all terms in $\lceil \sigma \rceil$ are closed. With Lemma 6.13 we conclude that $s\{\lceil \sigma \rceil\}^0 = \lceil p \rceil$ is closed. ∎

Theorem 6.15 (Soundness) If p is admissible and $p \succ_{\mathrm{LC}} q$, then $\lceil p \rceil \succ_{\mathrm{L}}^{\leq 1} \lceil q \rceil$

Proof By induction on $p \succ_{\mathrm{LC}} q$:

- The cases App and Var we directly conclude $\lceil p \rceil = \lceil q \rceil$.
- In the cases AppL and AppR we have $p = p_1{\cdot}p_2$ and $q = q_1{\cdot}q_2$. By the induction hypothesis, for one side i of $\lceil p \rceil = \lceil p_1 \rceil \lceil p_2 \rceil$ we have $\lceil p_i \rceil \succ_{\mathrm{L}}^{\leq 1} \lceil q_i \rceil$ and for the other side j, $p_j = q_j$. We conclude $\lceil p \rceil = \lceil p_1 \rceil \lceil p_2 \rceil \overset{\leq 1}{\Rightarrow} \lceil q_1 \rceil \lceil q_2 \rceil = \lceil q \rceil$.
- In the case β, we show that $\lceil \lambda s[\sigma] \rceil \lceil \lambda t[\tau] \rceil \succ_{\mathrm{L}} \lceil \lambda s[\lambda t[\tau]{::}\sigma] \rceil$. We note that $\lceil \lambda s[\sigma] \rceil \lceil \lambda t[\tau] \rceil = \lambda(s\{\lceil \sigma \rceil\}^1)\ \lambda(t\{\lceil \tau \rceil\}^1)$. This reduces to $(s\{\lceil \sigma \rceil\}^1)^0_{\lambda(t\{\lceil \tau \rceil\}^1)}$. By Lemma 6.12, that is equal to $\lceil \lambda s[\lambda t[\tau]{::}\sigma] \rceil$, thus the claim holds. ∎

Corollary 6.16 If p is admissible and $p \succ_{\mathrm{LC}}^* q$, then $\lceil p \rceil \succ_{\mathrm{L}}^* \lceil q \rceil$.

Proof Soundness combined with Fact 6.9. ∎

6.4 LC is Complete

Only reductions due to β affect the result of $\lceil\cdot\rceil$. The simplification steps due to App and Var vanish modulo $\lceil\cdot\rceil$. We therefore define the **simplification** $(\cdot)\wr\colon \mathrm{LC} \to \mathrm{LC}$ that performs as much simplifications as possible in admissible terms:

$$\begin{aligned} x\wr &:= x \\ (p{\cdot}q)\wr &:= p\wr{\cdot}q\wr \\ (x[\sigma])\wr &:= x\text{-th}_x\,\sigma \\ (st[\sigma])\wr &:= s[\sigma]\wr{\cdot}t[\sigma]\wr \\ (\lambda s[\sigma])\wr &:= \lambda s[\sigma] \end{aligned}$$

We call a term p **simplified** iff $p\wr = p$. Note that an admissible term is simplified iff it contains no Var- or App-redex.

From definition, we conclude three properties of simplification. They are proven by induction on the LC-term, whereby in the case of a closure, an additional induction on the body of the closure is needed:

Fact 6.17 $p \succ^*_{\mathrm{LC}} p\wr$

Fact 6.18 $\lceil p\wr\rceil = \lceil p\rceil$

Fact 6.19 If p is admissible, then $p\wr$ is simplified.

Note that we need admissibility here because all terms in all environments must be values.

Corollary 6.20 If p is admissible, then $p\wr$ is admissible as well.

Proof Fact 6.9 with Fact 6.17. ∎

Fact 6.21 If p is admissible and simplified and $p \succ_{\mathrm{LC}} q$, then $\lceil p\rceil \succ_{\mathrm{L}} \lceil q\rceil$.

Proof Induction on $p \succ_{\mathrm{LC}} q$. The cases for App and Var contradict the assumptions. The other cases follow with the corresponding rule from L. ∎

So simplification really performs every applicable simplification step, as every additional reduction would reduce the corresponding L-term.

Theorem 6.22 (Completeness) Assume an admissible p such that $\lceil p\rceil \succ_{\mathrm{L}} t$. Then there is q with $t = \lceil q\rceil$ and $p \succ^+_{\mathrm{LC}} q$

Proof We will prove the statement for simplified p. The general claim follows via Facts 6.17, 6.18, 6.19 and Corollary 6.20 (see Figure 6.1)

So out claim is that for any admissible, simplified p with $\lceil p\rceil \succ_{\mathrm{L}} t$, there is q with $t = \lceil q\rceil$ and $p \succ^+_{\mathrm{LC}} q$. We prove this by induction on p:

By admissibility, p is an application or a closure:

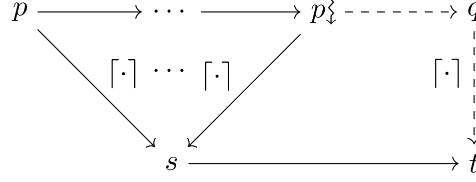


Figure 6.1: Sketch of Completeness Proof

- Assume $p = p_1{\cdot}p_2$ is an application. Case distinction on $\lceil p_1\rceil\,\lceil p_2\rceil \succ_{\mathrm{L}} t$. The cases for the recursive rules are straightforward by the induction hypothesis.

  In the other case, $\lceil p_1\rceil\,\lceil p_2\rceil \succ_{\mathrm{L}} t$ is a β-reduction. Thus there are s'_1, s'_2 such that $\lceil p_i\rceil = \lambda s'_i$ for $i = 1, 2$ and $t = {s'_1}^{0}_{\lambda s'_2}$.

  This means that $p_1 = \lambda s_1[\sigma_1]$ for suitable s_1 and σ_1. This follows by a case distinction on p_i: all other possible cases for p_i lead to a contradiction since the p_i are admissible and simplified with $\lceil p_i\rceil = \lambda s'_i$. By the same argument, there are s_2 and σ_2 with $p_2 = \lambda s_2[\sigma_2]$.

  Now the claim holds for $q = s_1[p_2{::}\sigma_1]$: First note that from $\lceil p_i\rceil = \lambda s'_i$, with $p_i = \lambda s_i[\sigma_i]$, we conclude $s'_i = s_i\{\lceil\sigma_i\rceil\}^1$ by Lemma 6.12. Together with Lemma 6.12 the first claimed property of q, namely $t = \lceil q\rceil$, holds. The other property $p \succ^{+}_{\mathrm{LC}} q$ follows by β-reduction.

- Assume $p = s[\sigma]$ is a closure. Since p is simplified, s must be an abstraction. Then $p{\downzigzagarrow}$ is an abstraction as well, and thus irreducible, contradicting $\lceil p\rceil \succ_{\mathrm{L}} t$. ∎

Note that without the restriction to simplified terms in the proof of Theorem 6.22, the prove would require an additional induction on the body of closures, where the case for L-application and LC-application are nearly identical. Introducing simplification straightens the proof and also shows more explicitly how $\succ_{\mathrm{LC}}$ refines $\succ_{\mathrm{L}}$ module $\lceil\cdot\rceil$ (see Figure 6.1).

We can also use simplification to connect normal forms of L and LC:

Corollary 6.23 For admissible p, $\lceil p\rceil$ is irreducible iff $p{\downzigzagarrow}$ is irreducible.

Proof We use Fact 6.21 (with Fact 6.18) to translate a reduction of $\lceil p\rceil = \lceil p{\downzigzagarrow}\rceil$ into a reduction of $p{\downzigzagarrow}$.

In the other direction, we use Theorem 6.22. ∎

Via induction, Theorem 6.22 translates to arbitrary reduction paths:

Corollary 6.24 If $\lceil p\rceil \succ^{*}_{\mathrm{L}} t$ with admissible p, then there is q with $t = \lceil q\rceil$ and $p \succ^{*}_{\mathrm{LC}} q$.

Combined with Corollary 6.23 and the soundness of $\mathcal{C}$, we conclude:

Fact 6.25 (Simulation) If p is admissible, then $\lceil p \rceil \Downarrow t$ iff there exists q such that $t = \lceil q \rceil$ and $p \Downarrow q$.

For some closed term s, taking $p = s[\,]$ in Fact 6.25 justifies that an interpreter for LC can be used to interpret L as well.

6.5 Evaluating LC

We now define a step-indexed interpreter $\text{eval}_{(\cdot)}(\cdot) : \mathbb{N} \to \text{LC} \to \text{option LC}$:

$$\begin{aligned}
\text{eval}_{k+1}(x[\sigma]) &= (x\text{-th}_x\, \sigma)_\top \\
\text{eval}_{k+1}(\lambda s[\sigma]) &= (\lambda s[\sigma])_\top \\
\text{eval}_{k+1}(st[\sigma]) &= \text{eval}_k(s[\sigma]\cdot t[\sigma]) \\
\text{eval}_{k+1}(x) &= x_\top \\
\text{eval}_{k+1}(\lambda s[\sigma]\cdot \lambda t[\tau]) &= \text{eval}_k(s[\lambda t[\tau]::\sigma]) \\
\text{eval}_{k+1}(p_1\cdot p_2) &= \text{eval}_k(q_1\cdot q_2) && \text{if } \text{eval}_k(p_i) = q_{i\top} \text{ for } i = 1,2 \\
\text{eval}_k(p) &= \bot && \text{otherwise}
\end{aligned}$$

Note that the first applicable equation shall be used.

Proposition 6.26 (Correctness) $p \Downarrow q$ iff $\text{eval}_k(p) = q_\top$ for some k.

Proof The direction for soundness is straightforward: Assuming $\text{eval}_k(p) = q_\top$, we can prove that there is a value q with $p \succ^*_{\text{LC}} q$ by induction on k, as each defining equation for the interpreter is sound.

For the completeness, we prove the stronger claim $p \Downarrow^k w \Rightarrow \exists k : \forall k' \geq k : \text{eval}_{k'}(p) = q$ by complete induction on k and case distinction on p.

The only interesting case is $p = p_1 \cdot p_2$ being an application: We use a helping lemma that allows to decompose the normalizing reduction of $p_1 p_2$ into three reductions; two normalizing ones for each of the p_i into p'_i and one that first contracts $p'_1 p'_2$ into q (by β) and then normalizes q, where the sum of the length of the the three normalizing reductions is $k - 1$ (since we isolated one β-reduction). This lemma is similar to Lemma 4.11.

As this decomposition corresponds to the computation performed by the step-indexed interpreter, the claim follows by using the induction hypothesis on the three decomposed reductions. ■

So combining the qualitative results of this chapter, we conclude that we can evaluate every L-term using LC:

Proposition 6.27 If $s \Downarrow t$, then there is k such that $t_\top = \lceil \text{eval}_k(s[\,]) \rceil$.

LC and a similar interpreter was used to prove that L can be simulated by a Turing machine in polynomial time. We conjecture that the time bound in [Rot15] can be even more improved, particularly that if $s \Downarrow^n t$, then LC can be used to evaluate s in time $O(n\,|s| + |t|)$.

We use a Coq-tactic based on Proposition 6.27 to evaluate L-terms. Note that to be more efficient, we use a lazy evaluation strategy in Coq to profit from hashconsing, a technique used in the evaluation of functional programming languages where common subterms are only stored once and referenced multiple times. Otherwise, the reduction $st[\sigma] \succ_{\mathrm{LC}} s[\sigma]{\cdot}t[\sigma]$ would copy the whole environment each time.

With this approach, we can evaluate terms like b_n from Example 6.1 significantly faster than with the naive approach: For example, b_{2500} normalizes in 0.01 instead of 15 seconds.

The use of the corresponding Coq-tactic reduced the compilation time of the whole Coq formalization in [For15] by a factor of ~ 3.5, from 7 to 2 minutes.

6.6 Related Systems

The relation between L and LC is similar to the one between $\lambda\beta$ and $\lambda\sigma$, a variant of $\lambda\beta$ with explicit substitution [ACCL91].

In both LC and $\lambda\sigma$, we have an explicit, syntactic representation of substitutions. But similar to L allowing for a simpler substitution than $\lambda\beta$ as described in Chapter 3, LC allows for a simpler explicit representation of substitutions than $\lambda\sigma$.

In [LM99], weak explicit substitution calculi similar to LC are studied. Especially the system 'weak explicit substitutions with ministeps semantics' has a fine grained semantics close to LC, but is not call-by-value. This requires the calculi in [LM99] to also allow the reduction of terms inside an explicit substitution, while for LC, we do not need allow this to have confluence.

Chapter 7

Coq Formalization

This thesis is accompanied[1] by a formalization in the formal proof management system Coq, compiled using version `8.5beta2 (August 2015)`.

The files are organized as follows:

File	**Specification**	**Proofs**	**Belongs to**
`UnifConfl.v`	35	58	Chapter 2
`L.v`	156	233	Chapter 3
`SKv.v`	165	200	Chapter 4
`SKvAbstraction.v`	141	373	Chapter 5
`LC.v`	180	295	Chapter 6
`LC_eval.v`	35	51	Section 6.5
In Total	712	1210	

Files with Number of Lines

Note that `L.v` is nearly completely taken over from [For14].

[1] available at `http://www.ps.uni-saarland.de/~kunze/bachelor.php`

Bibliography

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[AR15] Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape turing machines. *Theoretical Computer Science*, 603:23 – 42, 2015. Logic, Language, Information and Computation.

[Bar84] Henk Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[Bar97] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *The Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.

[For14] Yannick Forster. *A Formal and Constructive Theory of Computation.* Bachelor's Thesis, Saarland University, 2014. URL: `http://www.ps.uni-saarland.de/~forster/bachelor.php`.

[For15] Yannick Forster. *Verified Extraction from Coq to a Lambda-Calculus.* Research Immersion Lab, Saarland University, 2015. URL: `http://www.ps.uni-saarland.de/~forster/ri-lab.php`.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, 2008.

[LM99] Jean-Jacques Lévy and Luc Maranget. Explicit substitutions and programming languages. In *In 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 181–200. Springer, 1999.

[LM08] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[New42] Maxwell Newman. On theories with a combinatorial definition of "equivalence". *Annals of Math.*, 43(2):223–243, 1942.

[Nie00] Joachim Niehren. Uniform confluence in concurrent computation. *Journal of Functional Programming*, 10(5):453–499, sep 2000.

[Rot15] Marc Roth. *A reasonable time measure for the weak call-by-value lambda calculus*. Research Immersion Lab, Saarland University, 2015. URL: `http://www.ps.uni-saarland.de/~roth/ri-lab.php`.

[Sch24] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.