A Syntactic Theory of Sequential Control

by

Matthias Felleisen, Daniel P. Friedman,
Eugene Kohlbecker, & Bruce Duba

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 215

A Syntactic Theory of Sequential Control

by

M. Felleisen, D.P. Friedman, E. Kohlbecker, & B. Duba

February, 1987

This material is based on work supported in part by IBM Fellowship Awards to Matthias Felleisen and Eugene Kohlbecker, and by National Science Foundation grants number DCR 85-01277 and DCR 85-03325.
A SYNTACTIC THEORY OF SEQUENTIAL CONTROL

Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, Bruce Duba

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405, USA

Abstract

Sequential control operators like J and call/cc are often found in implementations of the \(\lambda\)-calculus as a programming language. Their semantics is always defined by the evaluation function of an abstract machine. We show that, given such a machine semantics, one can derive an algebraic extension of the \(\lambda_s\)-calculus. The extended calculus satisfies the diamond property and contains a Church-Rosser subcalculus. This underscores that the interpretation of control operators is to a certain degree independent of a specific evaluation strategy. We also prove a standardization theorem and use it to investigate the correspondence between the machine and the calculus. Together, the calculus and the rewriting machine form a syntactic theory of control, which provides a natural basis for reasoning about programs with non-functional control operators.

1. Deficiencies of the \(\lambda\)-calculus as a programming language

"The lambda calculus is a type-free theory about functions as rules, rather than graphs. ‘Functions as rules’ … refers to the process of going from argument to value, …" [1, p.3] No other words can better express why computer scientists have been intrigued with the \(\lambda\)-calculus. The rule character of function evaluation comes close to a programmer’s operational understanding of computer programs and, at the same time, the calculus provides an algebraic framework for reasoning about functions. Yet, this concurrence is also a major obstacle in the further development of the calculus as a programming language since it is based on simplicity rather than convenience.

The one and only means of computation in the calculus is the \(\beta\)-reduction rule which directly models function application. Although this suffices from a purist’s point of view, it is in many cases insufficient with respect to expressiveness and inefficient with respect to the evaluation process. For example, when a recursive program discovers the final result in the middle of the computation process, it should be allowed to immediately escape and report its value. Similarly, in an
erroneous situation a program must be able to terminate or to call an exception handler without delay. We could easily lengthen this list of examples, but the thrust is clear: functions-as-programs need more control over their evaluation.

The most general solution to the control problem within the functional realm originated in denotational semantics\(^1\) [4, 11, 19]. A program is evaluated by computing the value of its pieces and combining the results. When a particular component is being evaluated, one can think of the remaining sub-evaluations and the combination step as the rest of the computation or as the continuation of the current sub-evaluation. The crucial idea is to write programs in a style where functions can be used to simulate continuations. These programs always pass around and explicitly invoke (a functional representation of) the current continuation. They are thus able to direct the evaluation process: they may decide not to use the current continuation, to save it for later use, or to resume a continuation from some other point in time. However, such programs look clumsy and are hard to design. It is better to introduce linguistic facilities which give programs access to the current continuation when needed. Programs using these facilities are “much simpler, easier to understand (given a little practice) and easier to write. They are also more reliable since the machine carrying out the computations constructs the continuations mechanically…”\(^2\) Typical examples of such facilities in λ-calculus based languages are the J-operator [8, 9], label values [17], escape functions [16], call-with-current-continuation (abbreviated as call/cc) [15], and catch and throw [20].

Non-functional control operators “provide a way of pruning unnecessary computation and allow certain computations to be expressed by more compact and conceptually manageable programs.”\(^3\) If these operations make continuations

---

\(^1\) Indeed, the origin of the concept of a continuation can be traced back to A. van Wijngaarden who explained in a discussion at the IFIP Working Conference on Formal Language Description Languages, 1964 [18, p.24] that “this implementation of procedures is only so difficult because you have to take care of the goto statement. However, if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure return. That is the whole difficulty with procedure implementation. That’s why this is so simple; it’s exactly the same as a goto, only called in other words.”

\(^2\) C. Talcott about the introduction of note into Rum, a lexically-scoped dialect of Lisp [21, p.68].

\(^3\) C. Talcott wrote this remark in the context of escape mechanisms, but the spirit of her dissertation makes
available as first-class objects, as in Scheme or ISWIM, it is easy to imitate any
desired sequential control construct, e.g. escapes, error stops, search strategies as
applied in logic programming [7, 12], intelligent backtracking [5], and coroutining
[21]. Even though this is widely recognized, control operators are still regarded
with skepticism. Their addition seems rather ad hoc, because it only advances the
calculus as a programming language. On the algebraic side there are no rules re-
reflecting the new operations; proofs of program properties can no longer be carried
out in the syntactic domain. They must be based upon a semantic interpretation
in terms of abstract machines or denotational definitions [11, 21].

In this paper, an expanded revision of two preliminary reports [2, 3], we show
that the λ-calculus as an equational system can incorporate control operators and
that non-functional control may be characterized in a purely syntactic manner.
Starting from an idealized Scheme with an operational semantics, we design an
extended calculus-like system that captures the behavior of the additional control
operators. The theory is consistent in the sense that two different derivations start-
ing with the same term are confluent. Hence, it permits algebraic calculations in
the familiar style. A standardization theorem shows that the calculus defines a
programming language in its own right. Comparing the calculus and the program-
ming language semantics, we find that the two truly correspond to each other. The
standard computation function agrees with the evaluation function except for some
negligible differences and equality in the calculus implies indistinguishability on the
machine. Together with the machine semantics, the calculus provides a powerful
framework for reasoning about programs with non-functional control operators. We
exemplify and discuss this in more depth in the last section.

Since our base language has an applicative-order semantics, the resulting cal-
culus is an extension of Plotkin’s λ-value-calculus. Although assuming familiar-
ity with the conventional λ-calculus [1], we do not require understanding of Plotkin’s
variant [14]. Furthermore, we follow Plotkin’s plan [14] for the comparison of calculi
and programming languages, but, to keep the paper self-contained, we include some
appropriate explanations at key places.

2. Λc

The programming language Scheme [15] is the starting point of our language de-
sign. Of all the λ-calculus-based languages, its continuation accessing operation,
call/cc, is independent of the rest of the primitive language concepts. For example,

clear that it is also applicable to jump operations in general [21, p.16].
the J-operator [8] interferes with functional abstraction; the escape-construct [16] introduces a second binding facility. Thus, call/cc, which simply takes an argument and applies it to the current continuation, appears to be the ideal candidate. However, it too complicates the development of a calculus. When call/cc is used, it not only transfers control over the current continuation to the program, it also installs this continuation. This implies that a program has only partial control over the continuation.

To avoid these problems, we introduce a variant of call/cc into the λ-calculus term language. The next two subsections contain the extended syntax and its semantics. In the third one, we illustrate how to program with this new facility. We compare our example to a functional solution and briefly discuss the continuation-passing programming style as a generalization of the latter.

2.1. Syntax
The core of our programming language is the λ-calculus term set Λ. For the sake of simplicity, we concentrate on constant-free expressions. The extended language Λ_e includes two new types of applications: C- and A-applications. The formal definition of Λ_e is displayed in Definition 2.1. We adopt the notational conventions of the classical λ-calculus and write λxy.M for λx.λy.M, LMN for ((LM)N), and also CM for (CM), etc. where this is unambiguous.

The notion of free and bound variables in a term M carries over directly from the λ-calculus. Terms with no free variables are called closed terms or programs. We adopt Barendregt's convention of identifying terms (≡) that are equal except for some renaming of bound variables and his hygiene condition which says that in a discussion, free and bound variables are assumed to be distinct. Furthermore we extend Barendregt’s definition of the substitution function, M[x := N], to Λ_e in the natural way:

\[ x[x := N] \equiv N, \quad y[x := N] \equiv y \quad (x \neq y), \]
\[ (λy.M)[x := N] \equiv (λy.M[x := N]), \]
\[ (LM)[x := N] \equiv (L[x := N]M[x := N]) \]

and

\[ (CM)[x := N] \equiv (CM[x := N]), (AM)[x := N] \equiv (AM[x := N]). \]

2.2. Operational semantics
The intention behind the two operations C and A can be explained informally. An A-application represents an abort or stop operation, which terminates the program
Definition 2.1: The term sets \( \Lambda_c \) and \( \Lambda \)

The improper symbols are \( \lambda, (, ), , C, \) and \( A \). \( \text{Var} \) is a countable set of variables; the symbols \( x, k, f, v \), etc. range over \( \text{Var} \) as meta-variables but are also used as if they were elements of \( \text{Var} \). \( L, M, N, \ldots \) are meta-variables for \( \Lambda_c \). The term set \( \Lambda_c \) contains

- variables: \( x \) if \( x \in V \);
- abstractions: \( (\lambda x.M) \) if \( M \in \Lambda_c \) and \( x \in V \);
- applications: \( (MN) \) if \( M, N \in \Lambda_c \); \( M \) is called the function, \( N \) is called the argument;
- \( C \)-applications: \( (CM) \) if \( M \in \Lambda_c \), and \( M \) is called the \( C \)-argument;
- \( A \)-applications: \( (AM) \) if \( M \in \Lambda_c \), and \( M \) is called the \( A \)-argument.

The union of variables and abstractions is referred to as the set of values; \( U, V, \ldots \) are meta-variables for values.

\( \Lambda_c \), the term set of the traditional \( \lambda \)-calculus, stands for \( \Lambda_c \) restricted to variables, applications, and abstractions.

and returns the value of its argument. Whereas such an operation is commonly found in traditional languages, \( C \) and its relatives are only available in \( \lambda \)-calculus-based languages. The operation gives its argument complete control over the current continuation, that is, \( C \) applies its argument to an abstraction of what must be done in order to complete the program after evaluating the \( C \)-application. This step is also called labeling—or capturing—of continuations with reference to label values in more traditional languages. A continuation is invoked—or thrown to—by applying it to a value, just like a function. The \( C \)-operation and call/cc only differ in one point: call/cc implicitly invokes the current continuation on the value of its argument; \( C \) leaves this to its argument. \( C \) is equivalent to call/cc and \( A \) and vice versa:

\[
(CM) \equiv (\text{call/cc}(\lambda k. A(Mk))) \quad \text{and} \quad (\text{call/cc}M) \equiv (C\lambda k.k(Mk)).
\]

The formal semantics of \( \Lambda_c \) is defined via an abstract machine. We started from an operational interpretation of a denotational semantics in the style of Reynolds’s continuations-as-data structures interpreter [16]. Such a machine has three state
components: terms as control strings, environments for the evaluation of free variables, and continuation structures to remember the rest of a computation. The elimination of environments by quasi-substitution and the replacement of continuation codes by a special kind of context leads to an equivalent quasi-term rewriting system [2]. Since this rewriting system is more appropriate for the derivation of a calculus, we take it as the basis of our development.

The underlying term language $\Lambda_p$—see Definition 2.2—of the C-rewriting machine is an extension of $\Lambda_c$ with a new set of values: continuation points. A continuation point is a p-tagged applicative context. An applicative context is a term with a hole in it such that the path from the root of the term to the hole leads through applications only, and every subterm to the immediate left of the path is a value. We use $C[ ]$, $\ldots$ to range over applicative contexts, $C[M]$ to denote a term which is like the context $C[ ]$ but with $M$ put into the hole. Because the hole of an applicative context is never within an abstraction, the filling-in of a hole cannot capture free variables.

<table>
<thead>
<tr>
<th>Definition 2.2: The C-rewriting system</th>
</tr>
</thead>
<tbody>
<tr>
<td>The term language $\Lambda_p$:</td>
</tr>
<tr>
<td>$M ::= x \mid \langle p, C[ ] \rangle \mid \lambda x.M \mid MN \mid CM \mid AM$</td>
</tr>
<tr>
<td>$C[ ] ::= [ ] \mid V.C[ ] \mid C[ ]M.$</td>
</tr>
<tr>
<td>The C-transition function:</td>
</tr>
<tr>
<td>$C[(\lambda x.M)V] \xrightarrow{C} C[M[x := V]]$ (C1)</td>
</tr>
<tr>
<td>$C[CM] \xrightarrow{C} M\langle p, C[ ] \rangle$ (C2)</td>
</tr>
<tr>
<td>$C[\langle p, C_0[ ] \rangle V] \xrightarrow{C} C_0[V]$ (C3)</td>
</tr>
<tr>
<td>$C[AM] \xrightarrow{C} M.$ (C4)</td>
</tr>
</tbody>
</table>

The transition function of the C-rewriting system is displayed in Definition 2.2. It performs a single evaluation step on an entire program. The unique partitioning of a program into an applicative context and a C-redex determines the next rewriting
step. This completely orders evaluations on the C-rewriting machine; it implies that the evaluation of an application proceeds from left to right. The applicative context of a C-rewriting step also represents the current continuation. For the evaluation of a C-application the context is packaged into a continuation point, for an A-application it is thrown away. The invocation of a continuation point installs the associated control context, forgetting the current one. The evaluation function for programs is the transitive closure of the transition function:

\[ \text{eval}_C(M) = N \text{ if } M \xrightarrow{C}^* N \text{ such that } N \text{ is a value.} \]

From the transition rules we can deduce that \( \text{eval}_C \) is a partial function on programs: it either yields a value or it diverges. If \( \text{eval}_C \) yields a value for a program, we say that \( M \) has a value; we also generalize this to open expressions. The result is not necessarily in \( \Lambda_c \) since continuation points are a new kind of value; if it is in \( \Lambda_c \), we call it continuation-free.

The C-rewriting system defines an equivalence relation over \( \Lambda_c \). Intuitively two terms are operationally equivalent if one can replace the other in any program and the two resulting programs are indistinguishable with respect to \( \text{eval}_C \). The notion is due to J.H. Morris [13]; Plotkin [14] introduces the name “operational equivalence.” Its formalization depends on two concepts: program contexts and basic constants.

A program context is an arbitrary term with one hole. Unlike applicative contexts, it may capture free variables when filled with an open term. A set of values is referred to as a set of basic constants if it has a decidable equality predicate. Since \( \Lambda_c \) does not contain constant names, we arbitrarily pick the set of normal-form number representations in \( \Lambda_c \), e.g., the Church- or Turing-numerals [1]. With this in place, we can formulate

Definition 2.3. \( M, N \in \Lambda_c \) are operationally equivalent, \( M \simeq_C N \), iff for any program context \( C[\ ] \) such that \( C[MN] \) is closed, \( \text{eval}_C \) is undefined for both \( C[M] \) and \( C[N] \), or it is defined for both and if one of the programs yields a basic constant, then the value of the other is the same constant.

With an operational semantics à la SECD-machine [9] operational equivalence is rather opaque. Verifying the equivalence of two useful program pieces is almost impossible. With the C-rewriting system this becomes simpler. The transition function only uses one extraneous concept, namely applicative contexts, and this is naturally related to terms. However, the system only accounts for entire program
rewriting steps and these steps are completely ordered. There is no provision for local transformations, independent of a context, nor for a simultaneous reduction of redexes. In other words, the C-rewriting system is not quite the calculus-extension one would like for Λc. Before we consider this problem, we briefly illustrate programming in Λc and clarify its advantages.

2.3. Programming in Λc

Many traditional programming languages contain linguistic devices that can be programmed with continuation accessing operations like C. A typical example is the function exit facility. It permits a procedure to return a result immediately to its caller, avoiding recursive nestings. We demonstrate with a sample program how this is achieved with C. Yet, this is only a trivial use of continuations; for more interesting programs we refer the reader to the literature [6].

First we recall some common combinators and syntactic forms [1, 9]. The function I ≡ λx.x is the identity function; T ≡ λxy.x and F ≡ λxy.y stand for the truth values true and false, respectively. Given truth values, the implementation of the call-by-value version of the branching-form (if B M N) is B(λd.M)(λd.N)I where d is a dummy variable. We also adopt Barendregt’s numeral system for the λc-calculus where ‘n’ represents the number n, zero? is the 0-test predicate, and + denotes the addition function for numerals. For the example we assume that a tree is either empty or that it consists of a left-son tree, a number, and a right-son tree. The functions mt?, lson, rson, and num are the respective predicate and selector functions.

Given these definitions, consider the function Σ* which takes a tree of numbers and returns their sum:

\[
Σ^* ≡ Y_0(λs.λt. \\
\quad (if(\text{mt}\ t) '0' \\
\quad (+ (\text{num}\ t) (+ (\text{lson}\ t))(\text{rson}\ t))))
\]

where \( Y_0 ≡ λf.(λx.f(λz.xzz))(λx.f(λz.xzx)) \) represents the call-by-value recursion combinator [17]. Designing a purely functional program that immediately returns '0' upon encountering a tree element '0' is less trivial although this is only a minor modification to the original specification. With C, this new function Σ0* is a
straightforward extension of the \( \Sigma^* \)-function:

\[
\Sigma_0^* \equiv \lambda t. c \lambda k. k(Y_v(\lambda s. \lambda t.
\begin{align*}
& \text{if}(\text{mt}\, t)\, 0' \\
& \text{if}(\text{zero}\, (\text{num}\, t))\, (k'0') \\
& \quad (+ (\text{num}\, t)(+(s(\text{ison}\, t))(s(\text{rson}\, t)))))) \\
& \quad t).
\end{align*}
\]

For a comparison we give an \textit{intensionally equivalent} definition of this function in the \( \lambda \)-calculus. By this we mean that the following function is like \( \Sigma_0^* \) in that it performs a single-pass over the tree, adds numbers only if necessary and escapes as soon as a \( 0' \) is discovered:\footnote{There are alternatives to this version but they are all derivable from this function [22].}

\[
\lambda - \Sigma_0^* \equiv \lambda t. (Y_v(\lambda s. \lambda t k. \text{if}(\text{mt}\, t)\, (k'0')) \\
\quad \text{if}(\text{zero}\, (\text{num}\, t))\, 0' \\
\quad (s(\text{ison}\, t) \\
\quad (\lambda l. (s(\text{rson}\, t))(\lambda r. (k+(\text{num}\, t)(+l\, r))))))) \\
\quad t) I).
\]

The internal loop of this program simultaneously passes around the current tree and a function that can perform the rest of the \( \lambda - \Sigma_0^* \)-computation. Initially this simulated continuation is the identity function, indicating that \( \lambda - \Sigma_0^* \) need only return the result. At every junction of two subtrees, \( \lambda - \Sigma_0^* \) builds two continuations: \( (\lambda l. (s(\text{rson}\, t)\ldots)) \), which represents the application of the function to the right subtree, and \( \lambda r. k\ldots \), which combines the two partial results, passing them on to another continuation. When the function encounters an empty tree, it applies the simulated continuation to the intermediate result \( 0 \); when a \( 0 \) is discovered, the continuation function is thrown away so that no more tree nodes are visited.

The function \( \lambda - \Sigma_0^* \) is an optimized instance of a more general programming pattern: the continuation-passing style alluded to in the introduction. All \( \lambda \)-programs can be restructured into this style via the following well-known transformation (cps) [4, 14]:

\[
\begin{align*}
[x] & = \lambda k. kx, \\
[(\lambda x. M)] & = \lambda k. k(\lambda x. [M]), \\
[(M N)] & = \lambda k. [M](\lambda m. [N](\lambda n. m n k)).
\end{align*}
\]
To evaluate a cps'ed program, it must be applied to the initial continuation I. If the result of a program $M \in \Lambda$ is $\lambda x.N$, then $[M]\![I]$ yields $\lambda x.[N]$ [14]. For $\Lambda_c$, we must add two clauses:

$$[(CM)] = \lambda \kappa.[M](\lambda m.m(\lambda \nu.\kappa(\lambda \nu'.\nu'))I), \quad (cps4)$$
$$[(AM)] = \lambda \kappa.[M]I. \quad (cps5)$$

These equations reflect the formal definition of $C$ and $A$: $C$ transfers control over the current continuation to the program, continuing the evaluation as if it had just begun; $A$ throws away the current continuation.

The existence of the $[\cdot\!]$-morphism shows that $C$- and $A$-applications abstract from recurring programming patterns in a functional language. Theoretically, one can write programs in $\Lambda_c$ and translate them into $\Lambda$ for further manipulations [11], but this would defeat the purpose of abstraction. The correct solution is to extend the $\lambda_v$-calculus with axioms for these semantic abstractions. The need for these additional axioms is documented by the following two reformulations of (cps4) and (cps5):

$$[CM] = \lambda \kappa.(\lambda \kappa'.\kappa'((\lambda f.\kappa.((\lambda \nu.\kappa.(\lambda \nu'.\nu'))I))(\lambda f.[M](\lambda m.\nu.m\kappa))) \quad (cps4')$$
$$[AM] = \lambda \kappa.(\lambda \kappa'.\kappa'((\lambda \nu.\kappa.\nu'))(\lambda f.[M](\lambda m.\nu.m\kappa)). \quad (cps5')$$

The two equations treat $C$- and $A$-applications as if they were applications and as if $C$ and $A$ were ordinary abstractions in $\Lambda_c$ that map to the underlined parts, respectively. On the other hand, neither $C$ nor $A$ are the images of values in $\Lambda_c$ and therefore, ordinary $\beta$-reduction cannot support reasoning with $C$- or $A$-applications. New axioms must be developed.

3. The $\lambda_c$-calculus

The traditional $\lambda$-calculus may be perceived as an axiomatic theory as well as a reduction system. The two views are equivalent. The theory can only prove terms equal that are equal under the congruence relation generated from the $\beta$-reduction. From a computational viewpoint the reduction system is more attractive since it exposes the rule character of the calculus and its operational nature. Thus, it is quite natural when we go the inverse direction in this section, taking the specification of the $C$-rewriting rules as the point of departure and deriving the reduction system.

Once the calculus is derived, the next step is to investigate its fundamental properties. Among these, consistency and standardization are the most important.
The former means that equations in the calculus make sense, the latter implies that the calculus defines a programming language. Together, the two theorems provide a basis to tackle a correctness proof for the calculus.

3.1. Reductions and computations

A transition step in the C-rewriting system depends on partitioning the program into a C-redex and an applicative context. For the C1-step this dependency is superficial, for the others it is inherent because of the sequential nature of continuations. To establish a calculus, we must try to eliminate the context dependency as far as possible. In the case of (C1) this is trivial and yields the $\beta_v$-relation:

$$(\lambda x. M)N \overset{\beta_v}{\longrightarrow} M[x := N] \text{ provided that } N \text{ is a value.} \quad (\beta_v)$$

It completely captures (C1) and the underlying $\lambda_v$-calculus.

Next we consider $A$-applications. According to (C4), an $A$-application removes its applicative context. Case analysis of applicative contexts leads to appropriate notions of reduction. If an $A$-application $AM$ is within an applicative context $C[\ ]$ and to the left of some arbitrary term $N$, then first the $N$ must be thrown away and second, the rest of the context must be removed. This is a recursive problem: $C[\ ]$ can be eliminated in favor of $M$ by simply placing $AM$ in the hole. Thus, $C[(AM)N]$ should be related to $C[AM]$. Since this is independent of the applicative context, we can formulate our first notion of reduction for $A$-applications:

$$(AM)N \overset{A_L}{\longrightarrow} AM. \quad (A_L)$$

The second possible case, where $AN$ is to the right of a value $M$, is treated symmetrically:

$$M(AN) \overset{A_R}{\longrightarrow} AN \text{ provided that } M \text{ is a value.} \quad (A_R)$$

This covers all but the base case of applicative contexts.

The empty context requires special treatment. An occurrence of $AM$ at the root of a term must evaluate to $M$, but this cannot be a proper reduction. One can only apply this rule when the $A$-application is not embedded in a term. Otherwise the reduction system becomes inconsistent. Consider the expression $(AF)TT$. Applying the $A_L$-step twice results in $AF$; the top-level rule then leads to $F$. When the top-level relation is first applied to the embedded $A$-application, $(AF)TT$ goes to $FTT$ which in turn results in $T$. $T$ would thus equal $F$ and this is inconsistent. We
therefore introduce this top-level relation as a *computation rule* and denote it with a $\triangleright$ instead of the customary $\rightarrow$:

$$AM \triangleright A M.$$ \hfill (A_T)

When we complete the calculus later, we must add this computation rule at the right place.

The considerations for $C$-applications move along the same line. We must satisfy equations (C2) and (C3). Again, (C2) specifies that the context of a $C$-application must be removed. Thus, we expect that the $C$-reduction rules must be designed according to the position of $CM$ in an applicative context and that they must be similar to $A$-reductions. For example, the expression $(CM)N$ must relate to a term $CX$ for some term $X$.

For the correct design of $X$ we appeal to the intended semantics of the $C$-application. The $C$-application must capture the current continuation and supply it to its argument. Hence, if $X$ is the next $C$-argument, it will be applied to the continuation which stands for the rest of the context. This continuation must be passed on to the original $C$-argument $M$. Furthermore, $M$'s context includes an application with $N$ as the argument. In other words, if we let $f$ be the function which must be applied to $N$, then the continuation of $CM$ could be characterized by $\kappa(fN)$ where $\kappa$ stands for the continuation of $CX$. Since the continuation gets the function when it is invoked, it must be an abstraction whose parameter is $f$: $\lambda f.\kappa(fN)$. The term $X$, on the other hand, must be a function which accepts the continuation $\kappa$ and passes it on to the term $\lambda f.\kappa(fN)$. A first approximation of $X$ is $\lambda \kappa. M(\lambda f.\kappa(fN))$. This satisfies (C2) since it removes the context of a $C$-application and applies its argument to some encoding of the context, but continuation points also need to respect (C3).

The rewriting rule (C3) demands that, when a continuation is invoked, the current context is removed. This means for $\Lambda_c$-continuations that the first action must be an *abort* action to remove the current context. Hence, $\lambda f.A(\kappa(fN))$ is the correct continuation for $M$. The symmetric case where $CN$ is to the right of a value $M$ is treated in a similar way and so we define the two notions of reduction for the $C$-application

$$(CM)N \xrightarrow{CL} C\lambda \kappa. M(\lambda f.A(\kappa(fN)))$$ \hfill (C_L)

$$M(CN) \xrightarrow{CR} C\lambda \kappa. N(\lambda v.A(\kappa(Mv)))$$ \hfill (C_R)

provided that $M$ is a value.
We still need to consider the empty context, i.e., the occurrence of a C-application at the root of a term. The C-argument \( M \) must be applied to a function which simulates the continuation-point (\( p, [ ] \)). The natural choice is \( \lambda x.Ax \). Again, this relation is not a proper notion of reduction but a computation rule:

\[
CM \triangleright_C M(\lambda x.Ax).
\]  

\((C_T)\)

With this last rule we have derived the reduction and computation rules that are intuitively needed to simulate the C-transition function, but defining notions of reduction is only the first step. The next one is to build a one-step reduction relation. A one-step reduction relation is the extension of a notion of reduction to a relation which is compatible with the syntactic constructors. In other words, the extended relation connects terms which are the same except for two subterms related by a reduction rule. In our case four syntactic constructions must be considered: abstraction, application, C-application, and A-application. The two computation rules cannot be included in this relation since they are not applicable to nested subterms. Definition 3.1 contains a formal description of the one-step reduction relation \( \rightarrow_c \).

The final step in the development of a calculus is the construction of a congruence relation from the reduction relation, i.e., an equivalence relation which respects the syntactic constructors. Conforming to tradition, we do this in two stages: \( \rightarrow_c \) is the transitive-reflexive closure of \( \rightarrow_c \); its respective equivalence relation is \( =_c \). This, however, is not yet the final goal. We still need to build in the computation rules. Without computation rules it is impossible to find a standard computation function which simulates the machine evaluation: occurrences of C- and A-applications at the root of a term cannot be removed. We extend the reduction relation \( \rightarrow_c \) to a computation relation \( \triangleright_c \) by adding the top-level relations. Forming the symmetric, reflexive, and transitive closure of \( \triangleright_c \) results in an equivalence relation \( \equiv_c \) which establishes equality among terms according to reductions and computations. All these concepts are summarized in Definition 3.1.

The relation \( \equiv_c \) determines the \( \lambda_c \)-calculus and we write \( \lambda_c \vdash M \equiv_c N \) if the terms \( M \) and \( N \) are equal under \( \equiv_c \). This calculus is not traditional in the sense that it uses incompatible relations. The congruence relation \( =_c \) is somewhat weaker but more traditional and we consider it as a subcalculus. We also write \( \lambda_c \vdash M =_c N \) when we refer to proofs within the subcalculus.

### 3.2. Fundamental properties of the \( \lambda_c \)-calculus

The development of the basic notions of the \( \lambda_c \)-calculus raises a number of interest-
Definition 3.1: The $\Lambda_c$-calculus

Let $\overset{c}{\longrightarrow} := \overset{c}{C} \cup \overset{c}{C} \cup \overset{A}{A} \cup \overset{A}{A} \cup \overset{C}{C}$. The one-step $C$-reduction $\overset{c}{\longrightarrow}$ is the compatible closure of $\overset{c}{\rightarrow}$:

- $M \overset{c}{ightarrow} N \Rightarrow M \overset{c}{\rightarrow} N$;
- $M \overset{c}{\rightarrow} N \Rightarrow \lambda x.M \overset{c}{\rightarrow} \lambda x.N$;
- $M \overset{c}{\rightarrow} N \Rightarrow ZM \overset{c}{\rightarrow} ZN, MZ \overset{c}{\rightarrow} NZ$ for $Z \in \Lambda_c$;
- $M \overset{c}{\rightarrow} N \Rightarrow CM \overset{c}{\rightarrow} CN$;
- $M \overset{c}{\rightarrow} N \Rightarrow AM \overset{c}{\rightarrow} AN$.

The $C$-reduction is denoted by $\overset{c}{\longrightarrow}$ and is the transitive-reflexive closure of $\overset{c}{\rightarrow}$. We denote the smallest congruence relation generated by $\overset{c}{\rightarrow}$ with $\equiv_c$ and call it $C$-equality.

The $C$-computation $\triangleright_c$ is defined by: $\triangleright_c := \triangleright_c \cup \triangleright A \cup \overset{c}{\longrightarrow}$. The relation $\equiv_c$ is the smallest equivalence relation generated by $\triangleright_c$. We refer to it as computational $C$-equality.

ing questions. In principle, it makes sense to inspect every $\lambda$-calculus-theorem on its validity for the extended calculus. As mentioned at the outset of this section, we concentrate on the question of consistency and the existence of standard reduction and standard computation sequences. The central results of this subsection are captured in the consistency theorems 3.7 and 3.8, the definition of a standard computation function (Definition 3.9), and the standardization theorem 3.10; the reader may wish to concentrate on these points and to skip over the proof details in this subsection for the first reading.

The consistency problem is equivalent to proving the confluence of reduction and computation paths that start from the same place. In other words, we must prove a classical Church-Rosser theorem for reductions and a diamond theorem for computations. The proof of the Church-Rosser property for $\overset{c}{\rightarrow}$ is an application of Martin-Löf’s method for showing the corresponding result for $\overset{\beta}{\rightarrow}$. Since our presentation follows Barendregt’s rather closely, we only state the necessary lemmas and demonstrate some of the major modifications to the proofs.
First, we define a version of the parallel reduction relation \( \rightarrow^p \) for \( \rightarrow_c \). For the proof of the standardization theorem we also define a notion of the length or size of a parallel reduction: see Definition 3.2. Note that if \( M \) is a value and \( M \rightarrow^p N \) then \( N \) is a value.

The following lemma shows the relationship between \( \rightarrow_c \) and \( \rightarrow^p \). Its proof is obvious and omitted.

**Lemma 3.3.** \( \rightarrow_c C \rightarrow_c C \rightarrow C \rightarrow^p C \rightarrow_c C \).

Next we prove that in \( \rightarrow^p \) unlike in \( \rightarrow_c \) the expression \( M[x := N] \) reduces to \( M'[x := N'] \) in one step if \( M \) and \( N \) reduce to \( M' \) and \( N' \) in one step, respectively. However, for the proof of the standardization theorem we also need to know that this reduction is shorter than the one from \((\lambda x.M)N \) to \( M'[x := N'] \). The two proofs have the same structure and therefore—following Plotkin—we have merged them:

**Lemma 3.4.** Suppose \( M \rightarrow^p M' \), \( N \rightarrow^p N' \), and \( N \) is a value. Then the following statements hold:

(i) \( M[x := N] \rightarrow^p M'[x := N'] \)

(ii) \( s_R = s_{M[x := N]} \rightarrow M'[x := N'] \leq s_L = s_{(\lambda x.M)N} \rightarrow M'[x := N'] \)

**Proof.** The proof is a structural induction on the reduction \( M \rightarrow^p M' \). We have omitted the cases which are similar to the given ones.

(P1) \( M \rightarrow^p M' \equiv M \).

The result follows by induction on the structure of \( M \). We demonstrate it for the subcase \( M \equiv CP \equiv CP' \equiv M' \).

(i) \( (CP)[x := N] \equiv (CP[x := N]) \rightarrow (CP'[x := N']) \) if \( P \) is smaller than \( M \).

(ii)

\[
s_R = s_{P[x := N]} \rightarrow P'[x := N'] \\
\leq s_{P \rightarrow P'} + n(x, P')s_{N \rightarrow N'} \text{ by inductive hypothesis} \\
< n(x, P)s_{N \rightarrow N'} + 1 = s_L \text{ since } P' \equiv P.
\]

(P3) \( M \equiv (AP)Q \rightarrow^p M' \equiv AP' \) and \( P \rightarrow^p P' \).

(i) \( ((AP)Q)[x := N] \equiv (AP[x := N])Q[x := N] \rightarrow AP'[x := N'] \) by inductive hypothesis for \( P[x := N] \rightarrow P'[x := N'] \).
Definition 3.2: The parallel reduction $\parallel$.

The parallel reduction over $\Lambda$ is denoted by $\parallel$. $s_{M \rightarrow N}$ or just $s$ is the function which measures the size of the derivation $M \rightarrow N$. $n(x, M)$ is the number of free occurrences of $x$ in $M$.

(P1)  
$M \parallel M$,  
$s = 0$

(P2)  
$M \parallel M', N \rightarrow N'$,

$N$ is a value $\Rightarrow (\lambda x.M)N \rightarrow M'[x := N']$,

$s = s_{M \rightarrow M'} + n(x, M')s_{N \rightarrow N'} + 1$

(P3)  
$M \rightarrow M' \Rightarrow (AM)N \rightarrow (AM')$,

$s = s_{M \rightarrow M'} + 1$

(P4)  
$N \rightarrow N', M$ is a value $\Rightarrow M(AN) \rightarrow AN'$,

$s = s_{N \rightarrow N'} + 1$

(P5)  
$M \rightarrow M', N \rightarrow N' \Rightarrow (CM)N \rightarrow C\lambda \kappa.M'(\lambda f.A(\kappa(fN')))$,

$s = s_{M \rightarrow M'} + s_{N \rightarrow N'} + 1$

(P6)  
$M \rightarrow M', N \rightarrow N'$,

$M$ is a value $\Rightarrow M(CN) \rightarrow C\lambda \kappa.N'(\lambda v.A(\kappa(M'v)))$,

$s = s_{M \rightarrow M'} + s_{N \rightarrow N'} + 1$

(P7)  
$M \rightarrow N \Rightarrow \lambda x.M \rightarrow \lambda x.N$,

$s = s_{M \rightarrow N}$

(P8)  
$M \rightarrow N \Rightarrow CM \rightarrow CN$,

$s = s_{M \rightarrow N}$

(P9)  
$M \rightarrow N \Rightarrow AM \rightarrow AN$,

$s = s_{M \rightarrow N}$

(P10)  
$M \rightarrow M', N \rightarrow N' \Rightarrow MN \rightarrow M'N'$,

$s = s_{M \rightarrow M'} + s_{N \rightarrow N'}$
(P6) \( M \equiv P(CQ) \rightarrow P' \equiv C\lambda \kappa.C'Q'(\lambda v.A(\kappa(P'v))) \) and \( P \rightarrow P', Q \rightarrow Q'\), and \( P \) is a value.

(i) \((P(CQ))[x := N] \equiv P[x := N](CQ[x := N]) \rightarrow C\lambda \kappa.C'[x := N]'(\lambda v.A(\kappa(P'[x := N]'v)))\)

by inductive hypothesis for \( P \) and \( Q \) and the fact that \( P[x := N] \) is a value.

(ii)

\[
\begin{align*}
\sigma_R &= \sigma_P[x := N]\sigma[CQ[x := N]] - \sigma \lambda \kappa..Q'[x := N]'(\lambda v.A(\kappa(P'[x := N]'v))) \\
&= \sigma_P[x := N] + \sigma_{Q[x := N]} - \sigma_{Q'[x := N]}' + 1 \\
&\leq \sigma_P[x := N'] + n(x, P')\rightarrow N' + \sigma_{Q[x := N]} - \sigma_{Q'[x := N]}' + 1 \\
&\text{by inductive hypothesis for } \sigma_P[x := N'] \\
&\text{and } \sigma_{Q[x := N]} - \sigma_{Q'[x := N]}' \\
&< (\sigma_P + \sigma_{Q[x := N]}' + 1) + n(x, M')\rightarrow N' + 1 = \sigma_L. \\
\end{align*}
\]

(P9) \( M \equiv CP \rightarrow M' \equiv CP' \) and \( P \rightarrow P' \).

(i) \((CP)[x := N] \equiv CP[x := N] \rightarrow CP'[x := N]' \) by inductive hypothesis.

(ii)

\[
\begin{align*}
\sigma_R &= \sigma_P[x := N] - \sigma_{P[x := N]'} \\
&\leq \sigma_P[x := N'] + n(x, P')\rightarrow N' \text{ by inductive hypothesis} \\
&< \sigma_L. \quad \square
\end{align*}
\]

In addition to Lemma 3.4 we must show that two contractums of \( C_L \)- or \( C_R \)-
redexes reduce to each other in one \( \rightarrow \)-step if the respective subterms do. Again,
the second and fourth claim of the following lemma are actually needed for the standardization theorem:

**Lemma 3.5.** Suppose \( M \rightarrow M' \) and \( N \rightarrow N' \). Then the following statements hold:

(i) \( C\lambda \kappa.M(\lambda f.A(\kappa(fN))) \rightarrow C\lambda \kappa.M'(\lambda f.A(\kappa(fN'))) \)
(ii) \( s_R < s_L \) where

\[
S_R = S_{\lambda \kappa. M(\lambda f. A(\kappa(fN)))} \rightarrow \lambda \kappa. M'(\lambda f. A(\kappa(fN')))
\]

\[
S_L = S_{(CM)N} \rightarrow \lambda \kappa. M'(\lambda f. A(\kappa(fN'))).
\]

And if \( M \) is a value then

(iii) \( \lambda \kappa. N(\lambda v. A(\kappa(Mv))) \rightarrow \lambda \kappa. N'(\lambda v. A(\kappa(M'v))) \)

(iv) \( s_R < s_L \) where

\[
S_R = S_{\lambda \kappa. N(\lambda v. A(\kappa(Mv)))} \rightarrow \lambda \kappa. N'(\lambda v. A(\kappa(M'v)))
\]

\[
S_L = S_{(CM)N} \rightarrow \lambda \kappa. N'(\lambda v. A(\kappa(M'v))).
\]

**Proof.** We show (i) and (ii) by straightforward calculations:

(i) \( N \rightarrow N' \) hence \( \lambda f. A(\kappa(fN)) \rightarrow \lambda f. A(\kappa(fN')) \),

\( M \rightarrow M' \) hence \( M'(\lambda f. A(\kappa(fN))) \rightarrow M'(\lambda f. A(\kappa(fN'))), \)

and \( \lambda \kappa. (M'(\lambda f. A(\kappa(fN)))) \rightarrow \lambda \kappa. (M'(\lambda f. A(\kappa(fN')))), \)

and therefore \( \lambda \kappa. M(\lambda f. A(\kappa(fN))) \rightarrow \lambda \kappa. M'(\lambda f. A(\kappa(fN'))). \)

(ii) \( s_R = s_{M \rightarrow M'} + s_{N \rightarrow N'} < s_{M \rightarrow M'} + s_{N \rightarrow N'} + 1 = s_L \).

The proofs of propositions (iii) and (iv) follow the same pattern. □

Everything is in place to state and prove the diamond lemma for the parallel reduction:

**Lemma 3.6.** The relation \( \rightarrow \) satisfies the diamond property, i.e., if \( M \rightarrow L_i \) then there exists an \( N \) such that \( L_i \rightarrow N \) for \( i = 1, 2 \).

**Proof.** Again, the proof is an induction on the structure of the reduction \( M \rightarrow \) \( L_i \). We only discuss two cases. The rest of the possible cases are either similar to some of the presented ones or can be found in Barendregt’s corresponding proof.

(P6) \( M \equiv P(\lambda \kappa. Q_1(\lambda v. A(\kappa(P_1v)))) \) and \( P \rightarrow P_1, Q \rightarrow Q_1 \),

and \( P \) is a value.

There are two possible cases for the reduction from \( M \) to \( L_2 \) since \( P \) is a value and \( \lambda \kappa. Q \) is not:

a) \( L_2 \equiv P_2(\lambda \kappa. Q_2) \) and \( P \rightarrow P_2, Q \rightarrow Q_2 \).

Note that \( P_2 \) is a value. An application of Lemma 3.5 yields \( N \equiv \lambda \kappa. Q_3(\lambda v. A(\kappa(P_3v))) \) where \( P_3 \) and \( Q_3 \) are the terms which must exist for \( P \rightarrow P_1 \) and \( Q \rightarrow Q_1 \), for \( i = 1, 2 \) according to the inductive hypothesis.
b) \( L_2 \equiv \mathcal{C} \lambda \kappa. Q_2(\lambda v. A(\kappa(P_2v))) \) and \( P \rightarrow_1 P_2, Q \rightarrow_1 Q_2 \).

Again, an application of Lemma 3.5 and of the inductive hypothesis for \( P \rightarrow_1 P_1, Q \rightarrow_1 Q_1, i = 1, 2 \) produces terms \( P_3, Q_3 \) such that \( \mathcal{N} \equiv \mathcal{C} \lambda \kappa. Q_3(\lambda v. A(\kappa(P_3v))) \).

(P10) \( M \equiv PQ \rightarrow_1 L_1 \equiv P_1Q_1 \) and \( P \rightarrow_1 P_1, Q \rightarrow_1 Q_1 \).

This time we have to distinguish 6 possible subcases:

a) \( L_2 \equiv R_2[x := Q_2] \) and \( P \equiv \lambda x. R, R \rightarrow R_2, Q \rightarrow Q_2 \), and \( Q \) is a value. But then \( P_1 \equiv \lambda x. R_1, R \rightarrow R_1 \), and \( Q_1 \) is a value. By inductive hypothesis we must be able to find \( R_3 \) and \( Q_3 \) such that with Lemma 3.4 \( \mathcal{N} \equiv R_3[x := Q_3] \).

b) \( L_2 \equiv AR_2 \) and \( P \equiv AR, R \rightarrow R_2 \).

Again, \( P_1 \equiv AR_1 \) and \( R \rightarrow R_1 \). By inductive hypothesis we can find an \( R_3 \) such that \( \mathcal{N} \equiv AR_3 \).

c) \( L_2 \equiv AR_2 \) and \( Q \equiv AR, R \rightarrow R_2 \), and \( P \) is a value.

This case is like b).

d) \( L_2 \equiv \mathcal{C} \lambda \kappa. R_2(\lambda f. A(\kappa(fQ_2))) \) and \( P \equiv CR, Q \rightarrow Q_2, R \rightarrow R_2 \).

This implies that \( P_1 \equiv CR_1, R \rightarrow R_1 \). An application of the inductive hypothesis and Lemma 3.5 shows that \( \mathcal{N} \equiv \mathcal{C} \lambda \kappa. R_3(\lambda f. A(\kappa(fQ_3))) \).

e) \( L_2 \equiv \mathcal{C} \lambda \kappa. R_2(\lambda v. A(\kappa(P_2v))) \) and \( Q \equiv CR, P \rightarrow P_2, R \rightarrow R_2 \), and \( P \) is a value.

This case is like d).

f) \( L_2 \equiv P_2Q_2 \). Trivial. □

Putting things together we get the Church-Rosser property for \( \rightarrow_c \):

**Theorem 3.7.** The relation \( \rightarrow_c \) is Church-Rosser.

**Proof.** \( \rightarrow_c \) is the transitive closure of \( \rightarrow_1 \). Since \( \rightarrow_1 \) satisfies the diamond property so does \( \rightarrow_c \). □

An alternative proof of the above theorem is based on the Hindley-Rosen method for showing that the \( \beta\eta \)-reduction is CR. This requires checking that each reduction is CR and that they commute with each other. In our case the second part would be laborious since five (!) different rules are involved. The above proof also has the advantage that it neatly ties in with the proof of the standardization theorem in the second half of this section.

Based on Theorem 3.7 we can show that \( \triangleright_c \) satisfies the diamond property which is sufficient to establish consistency:
Theorem 3.8 (Consistency). The relation $\triangleright_e$ satisfies the diamond property.

Proof. Assuming that $M \triangleright_e L_i$ for $i = 1, 2$, we need to show that there exists a term $N$ such that $L_i \triangleright_e N$. We proceed by a case analysis on $M \triangleright_e L_1$.

$(A_T)$ $M \triangleright_A L_1$ and $M \equiv (AL_1)$.
Then there are only two possible cases for the step from $M$ to $L_2$:

a) $M \rightarrow_e (AK_2)$.
But then $L_1 \rightarrow_e K_2$ and we can take $N \equiv K_2$.

b) $M \triangleright_A L_2$.
Trivially, $M \equiv (AL_1) \equiv (AL_2)$ and $N \equiv L_1 \equiv L_2$.

$(C_T)$ $M \triangleright_C L_1$ and $M \equiv (CL_1)$. This case is just like $(A_T)$.

$(\rightarrow_e) M \rightarrow_e L_1$.
Three cases are possible. Two of them are symmetric to the previous ones.
The third one is $M \rightarrow_e L_2$, but then we just apply the Church-Rosser Theorem for $\rightarrow_e$. □

The theorem establishes the following traditional corollary:

**Corollary.** If $M \equiv_e N$ then there exists an $L$ such that $M \triangleright_e^* L$ and $N \triangleright_e^* L$.

With the Church-Rosser theorem in place, we can tackle the standardization theorem. A standard reduction sequence for the $\lambda$-calculus is usually defined with respect to the position of redexes within a term and their residuals. Plotkin gives an equivalent, but more elegant and intuitive definition. It requires the notion of a standard reduction function which reduces the first—top-down and left-to-right—redex in a $\lambda$-term not inside an abstraction. This function is then extended to standard reduction sequences by forming something like a compatible closure. The extended syntax and the computation rules in the $\lambda_e$-calculus require a slightly more complex construction. In particular, the constructions of a standard reduction function and a standard reduction sequence must proceed in two stages such that computation rules do not interfere with reductions. Nevertheless, the definitions remain intuitive and are formalized in Definition 3.9.

The theorem which we want to prove can now be stated as:

**Theorem 3.10 (Standardization).** $M \triangleright_e^* N$ if and only if there exists a $C^e$-SCS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N$.

The proof is divided into two parts. First, we show that there is a standardization theorem for reductions. Second, we give a method for reshuffling $\triangleright_e$-
Definition 3.9: Standard reduction sequences and functions

The standard reduction function, denoted by $\rightarrow_{sc}$, is defined by:

$M \xrightarrow{c} N \Rightarrow M \xrightarrow{sc} N$;

$M \xrightarrow{sc} M' \Rightarrow MN \xrightarrow{sc} M'N$;

$M$ is a value, $N \xrightarrow{sc} N' \Rightarrow MN \xrightarrow{sc} MN'$.

Standard reduction sequences, abbreviated C-SRS-s, are defined by:

$x \in V \Rightarrow x$ is a C-SRS;

$N_1, \ldots, N_k$ is a C-SRS $\Rightarrow$

$\lambda x.N_1, \ldots, \lambda x.N_k, \mathcal{C}N_1, \ldots, \mathcal{C}N_k$, and $\mathcal{A}N_1, \ldots, \mathcal{A}N_k$ are C-SRS-s;

$M \xrightarrow{sc} N_1$, and $N_1, \ldots, N_k$ is a C-SRS $\Rightarrow M, N_1, \ldots, N_k$ is a C-SRS

$M_i, \ldots, M_j$ and $N_1, \ldots, N_k$ are C-SRS-s $\Rightarrow$

$M_i N_1, \ldots, M_j N_1, \ldots, M_j N_k$ is a C-SRS.

The standard computation function for $\lambda_c$ extends $\rightarrow_{sc}$ to computations:

$\xrightarrow{sc} = \Rightarrow \cup \mathcal{C} \cup \mathcal{A} \cup \rightarrow_{sc}$.

Standard computation sequences, $\mathcal{C}^p$-SCS-s, are defined by:

$N_1, \ldots, N_k$ is a C-SRS $\Rightarrow N_1, \ldots, N_k$ is a $\mathcal{C}^p$-SCS;

$M \xrightarrow{sc} N_1$ and $N_1, \ldots, N_k$ is a $\mathcal{C}^p$-SCS $\Rightarrow M, N_1, \ldots, N_k$ is a $\mathcal{C}^p$-SCS.

The notation $\xrightarrow{sc}^+$ and $\xrightarrow{sc}^*$ stand for the transitive and transitive-reflexive closure of $\xrightarrow{sc}$, respectively; $\xrightarrow{sc}^i$ indicates $i$ applications of $\xrightarrow{sc}$.

computation sequences into $\mathcal{C}^p$-SCS-s. The method is based on the first standardization theorem and it utilizes the consequence of Theorem 3.8 that at the root of the term computations and reductions are interchangable.

The standardization theorem for the reductions is:

**Theorem 3.11.** $M \xrightarrow{c} N$ if and only if there is a C-SRS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N$. 
Proof. The direction from right to left is trivial. For the opposite we follow Plotkin’s plan for the corresponding theorem about the $\lambda_v$-calculus. First, the sequence of $\rightarrow_{sc}$-steps is replaced by a sequence of steps using the parallel reduction $\rightarrow_p$. This follows from Lemma 3.3. Then we show with the following lemma that one can recursively transform the resulting sequence of $\rightarrow_p$ reductions into a C-SRS.

Lemma 3.12. If $M \rightarrow N_1$ and $N_1, \ldots, N_j$ is a C-SRS then there exists a C-SRS $L_1, \ldots, L_n$ with $M \equiv L_1$ and $L_n \equiv N_j$.

Proof. The proof is a lexicographic induction on $j$, on the size of the proof $M \rightarrow N_1$, and on the structure of $M$. We proceed by case analysis on the last step in $M \rightarrow N_1$ and omit all the cases which are similar to the presented ones or which are treated by Plotkin:

(P3) $M \equiv (AP)Q \rightarrow N_1 \equiv AP_1$ and $P \rightarrow P_1$.

But then we also have $M \rightarrow_{sc} AP$ and $AP \rightarrow AP_1$ by a proof which is shorter than the proof $M \rightarrow N_1$. Hence, by inductive hypothesis we find a C-SRS from $AP$ to $N_j$ and can then build the required C-SRS from $M$ to $N_j$.

(P6) $M \equiv P(CQ) \rightarrow N_1 \equiv Cl \kappa. Q_1(\lambda v. A(\kappa(P_1v)))$ and $P \rightarrow P_1, Q \rightarrow Q_1$, and $P$ is a value.

Again, $M$ can immediately be reduced to $Cl \kappa. Q(\lambda v. A(\kappa(Pv)))$ by $\rightarrow_{sc}$. By Lemma 3.4 we know that

$$Cl \kappa. Q(\lambda v. (\kappa(Pv))) \rightarrow Cl \kappa. Q_1(\lambda v. A(\kappa(P_1v)))$$

by a proof that is shorter than the one for $M \rightarrow N_1$. Therefore, by inductive hypothesis, we can find a C-SRS from $Cl \kappa. Q_1(\lambda v. A(\kappa(P_1 v)))$ to $N_j$ from which we build the required C-SRS from $M$ to $N_j$.

(P9) $M \equiv CP \rightarrow M' \equiv CP'$ and $P \rightarrow P'$.

Here $N_i \equiv CN'_i$ for all $i, 1 \leq i \leq j$. Now consider $P \rightarrow N'_1, \ldots, N'_j$. $P$ is obviously smaller than $M$ and we can apply the inductive hypothesis to find a C-SRS from $P$ to $N'_j$. Wrapping every element of this sequence in a C-application yields the required reduction sequence.

(P10) This case does not differ from Plotkin’s corresponding case but it requires that $M \rightarrow N \rightarrow_{sc} L$ can be transformed into $M \rightarrow_{sc} K \rightarrow L$ for some appropriate term $K$. This is proven in a separate lemma. □
The next lemma shows that \( \rightarrow \) and \( \rightarrow_{sc} \) commute as required by case (P10) of the preceding lemma:

Lemma 3.13. If \( M \rightarrow M' \rightarrow_{sc} M'' \) then there exists an \( L \) such that \( M \rightarrow_{sc}^+ L \rightarrow M'' \).

Proof. Plotkin’s proof of his Lemma 8, section IV goes through with almost no change. It is a lexicographic induction on the size of the reduction \( M \rightarrow M' \) and of \( M \). It is divided according to the last parallel reduction step. Cases (P1) through (P9) are routine with (P5) and (P6) relying on Lemma 3.5. The last case again induces the need for another lemma and deserves some explanation.

For case (P10) we assume that \( M = PQ \rightarrow M' \equiv P'Q' \) because \( P \rightarrow P' \), \( Q \rightarrow Q' \), and \( M' \rightarrow_{sc} M'' \). We now proceed by an analysis on this standard reduction step and consider two typical subcases:

a) \( M' \equiv (AP'_1)Q' \rightarrow_{sc} M'' \equiv AP'_1 \).

So we have \( P \rightarrow AP'_1 \). But then we claim that there exists an \( L \) such that \( P \rightarrow_{sc}^* L \rightarrow AP'_1 \) and \( L \) is an \( A \)-application. With the rules for \( \rightarrow_{sc} \) we get that \( (PQ) \rightarrow_{sc}^* (LQ) \rightarrow_{sc} L \rightarrow AP'_1 \equiv M'' \).

b) \( M' \equiv (CP'_1)Q' \rightarrow_{sc} M'' \equiv C\lambda x.P'_1(\lambda f.A(\kappa(fQ'))) \).

We proceed just like in a). Given that \( P \rightarrow CP'_1 \) we again claim that there is an \( K \) such that \( P \rightarrow_{sc}^* K \rightarrow CP'_1 \) and \( K \) is a \( C \)-application. \( K \) and \( Q' \) form the required \( L \) in the obvious way and the rest is similar to a). \( \square \)

There are four propositions left that we have claimed or that we need through our adoption of Plotkin’s proofs. All the necessary proofs are quite straightforward, but for the sake of completeness we state the lemmas:

Lemma. If \( M \rightarrow (AN) \) where \( M \) is an application then there exists an \( L \) which is an \( A \)-application and \( M \rightarrow_{sc}^+ L \rightarrow (AN) \).

Lemma. If \( M \rightarrow (CN) \) where \( M \) is an application then there exists an \( L \) which is a \( C \)-application and \( M \rightarrow_{sc}^+ L \rightarrow (CN) \).

Lemma. If \( M \rightarrow (\lambda x.N) \) where \( M \) is an application then there exists an \( L \) which is an abstraction and \( M \rightarrow_{sc}^+ L \rightarrow (\lambda x.N) \).

Lemma. If \( M \rightarrow x \) where \( x \) is a variable then \( M \rightarrow_{sc}^+ x \).

Equipped with this first standardization theorem for reductions, it is easy to finish the proof of Theorem 3.10, but before, we need to clarify one more fact:
Lemma 3.14. If \( N_1, \ldots, N_k \) is a C-SRS where \( N_k \equiv \mathcal{C}L \) or \( N_k \equiv \mathcal{A}L \) then there exists a \( j, 1 \leq j < k \) such that for all \( i, 1 \leq i < j, N_i \rightarrow_{sc} N_{i+1} \), and for all \( i, j \leq i < k, N_i \rightarrow_c N_{i+1} \) and \( N_i \not\rightarrow_{sc} N_{i+1} \).

Proof. A straightforward induction on \( k \). □

And finally, here is the proof of the main result of this section:

Proof of Theorem 3.10. The proof is an induction on the number of computations, \( \triangleright_c \) and \( \triangleright_A \), used in the evaluation of \( M \) to \( N \). If there are no computations involved, we can form the C-SRS for reducing \( M \) to \( N \) and we have the desired result. Now suppose there is at least one reduction of type \( \triangleright_A \). Then we have the following situation:

\[
M \equiv M_1 \rightarrow_c \ldots \rightarrow_c M_k \equiv AM_i \triangleright_A M_{k+1} \equiv M_i \rightarrow_c \ldots \rightarrow_c M_n \equiv N.
\]

By forming the C-SRS for the reduction from \( M_1 \) to \( M_k \) we get by Theorem 3.11 and Lemma 3.14:

\[
M \equiv M_1 \rightarrow_{sc} \ldots \rightarrow_{sc} M_i \equiv AM_i \rightarrow_c \ldots \rightarrow_c M_k \equiv AM_i \\
\triangleright_A M_{k+1} \equiv M_i \rightarrow_c \ldots \rightarrow_c M_n \equiv N.
\]

Since \( \rightarrow_c \) and \( \triangleright_A \) are interchangable at the root of a term—see Theorem 3.8—we can move the computation forward:

\[
M \equiv M_1 \rightarrow_{sc} \ldots \rightarrow_{sc} M_i \equiv AM_i \triangleright_A M_i \rightarrow_c \ldots \rightarrow_c M_i \rightarrow_c \ldots \rightarrow_c M_n \equiv N.
\]

By inductive hypothesis we get a similar reduction sequence for the reduction from \( M_i \) to \( N \). Since \( M \rightarrow_{sc} M_i \) we can form the desired \( C^p \)-SCS from \( M \) to \( N \).

The case of \( \triangleright_C \)-computations is treated similarly. □

This ends the investigation of the logical properties of the \( \lambda_c \)-calculus. The two results enable us to show that the \( \lambda_c \)-calculus corresponds to the C-rewriting system.

4. The machine-calculus correspondence

Following Plotkin [14] a calculus is correct with respect to a programming language if the operational semantics of the calculus agrees with the original machine semantics and if equality in the calculus implies operational equality. If a calculus is correct, we say that it corresponds to the machine, thus underlining that calculi and programming languages are pairs which determine each other.
To prove the equivalence of the machine semantics with the operational rewriting semantics of $\lambda_c$, we show that the standard computation function simulates the rules (C1) through (C4) of Definition 2.2. Since an evaluation may return a continuation point as (part of) the result, we first construct a morphism from $\lambda_p$-terms to $\lambda_c$-terms so that the $C$-rewriting machine can be properly unloaded. The major task of this morphism is to encode contexts as terms in the same way as the $C$-reductions do.

For the construction of the morphism it is advantageous to look at contexts from the inside out, i.e.,


The empty context in a continuation point means that the continuation was captured with a $C$-application at the root of the term. Hence, $[\ ]$ maps to $\lambda x. Ax$. If the hole is to the left of some arbitrary term $M$ in some context $C[\ ]$, then a $C$-application would use $C_L$ to construct the next piece of the continuation. This new piece would look like $\lambda f.A(\kappa(fM))$ where $\kappa$ stands for the encoding of $C[\ ]$ and so we are led to the following definition of the morphism $[\cdot]_c$ from contexts to terms:

$$[[\ ]\!_c\! \equiv \lambda x.Ax$$

$$[[C[\ ]M]\!_c\! \equiv \lambda f.A([C[\ ]\!_c\! (fM))$$

$$[[C[V[\ ]]\!_c\! \equiv \lambda v.A([C[\ ]\!_c\! (Vv)).$$

The map from $Q$ to $\overline{Q}$ replaces continuation points in $Q$ by terms:

$$\langle p, C[\ ] \rangle \equiv [C[\ ]]_c, \ p \equiv x, \ \lambda x.M \equiv \lambda x.M, \ MN \equiv MN, \ cM \equiv cM, \ \lambda M \equiv \lambda M.$$

Given the morphisms, we can attempt to prove a simulation theorem for the four $C$-rewriting clauses. The $\beta_c$-step, i.e. (C1), and steps (C2) and (C4) are clearly reflected in the definition of the standard computation function. In particular the latter two rules were a major guide in the derivation of the reduction system and the map $[\cdot]_c$ was designed according to the resulting notions of reduction:

**Lemma 4.1.** For any applicative context $C[\ ]$,

(i) $C[CM] \rightarrow^{s_c} M[C[\ ]_c]$, and

(ii) $C[AM] \rightarrow^{s_c} AM \rightarrow^{s_c} M$.

**Proof.** The proof is a straightforward induction on the structure of applicative contexts. $\Box$
We are, however, unable to show that the standard reduction function satisfies rule (C3). This transition rule requires that a continuation invocation removes the current context and that it continues as if the old context—filled with the argument—were the new term. The first condition is clearly implemented since continuations immediately perform an \( \Lambda \)-application. The second one causes problems. In the \( \lambda_c \)-calculus continuations are constructed to simulate the behavior of contexts, but in the machine continuations are contexts. Thus, when a continuation is to be captured after another one was invoked, the transition in the machine and the one via the standard computation function diverge. The machine simply labels the current context which contains the old continuation context; the standard computation sequence encodes for a second time the term which simulates the former continuation.

The nature of the problem is best illustrated with an example. Suppose the continuation point \( (p, C[[V]]) \) is invoked on the value \( F \):

\[
(p, C[[V]])F \xrightarrow{C} C[FV].
\]

Furthermore, assume that the application \( FV \) evaluates to \( D[CP] \) after some \( \beta \)-steps. Then the \( C \)-transition reaches the term \( P(p, C[D[ ]]) \). According to Lemma 4.1, if \( K_c \equiv [C[ ]]_c \), the corresponding reduction sequence in the \( \lambda_c \)-calculus begins with:

\[
[C[[V]]]_c F \xrightarrow{\beta_{sc}^+} K_c(FV).
\]

The next few \( \beta \)-steps for \( FV \) are correctly simulated by the standard computation function:

\[
K_c(FV) \xrightarrow{\beta_{sc}^+} K_cD[CP].
\]

This last term also constructs a continuation—just like \( C[D[CP]] \)—but the continuation encodes the term \( K_c \) instead of the context \( C[ ] \):

\[
K_cD[CP] \xrightarrow{\beta_{sc}^+} P[K_cD[ ]}_c.
\]

Clearly, \([C[D[ ]]]_c \) is not equal to \([K_cD[ ]]_c \) and thus, a naïve version of the simulation theorem fails. The best we can hope for is that the standard computation simulation of the \( C \)-transition function preserves a relation between continuation points and terms.

From the above lemma and the example one may suspect that a continuation point like \( (p, C[D[ ]] \) is related to the terms \([C[D[ ]]]_c \) and \([C[ ]]_cD[ ]_c \). However, the situation in our example could recur many times. Instead of having two
contexts composing a new one, we would then have several of them. In fact, we
must account for all possible finite decompositions of a given context into smaller
contexts, including the empty one. Each context can be encoded as a term by it-
self; each of these encoded contexts can be a part of a bigger context which is being
coded. We have formalized this relation in Definition 4.2.

**Definition 4.2:** The continuation point-term correspondence

| The relation \( \mathord{\approx}_p \) basically compares continuation points in \( \Lambda_p \) to terms in \( \Lambda_c \). It is
| defined inductively over \( \Lambda_p \) and applicative contexts (over \( \Lambda_p \)):
| \( \langle p, C[ ] \rangle \approx_p \ldots \langle [C_1[ ],\ldots,C_n[ ]],\ldots,C_n[ ] \rangle \)
| for all \( n \) such that \( C[ ] \equiv C_1[\ldots C_n[ \ldots \ldots ]\ldots] \) and \( C_i[ ] \approx_p C_i[ ] \) for all \( i \leq n \);
| \( x \approx_p x, \lambda x.P \approx_p \lambda x.\overline{P}, P \approx_P Q, C.P \approx_p CP, AP \approx_p AP \) if \( P \approx_p P \) and \( Q \approx_p Q \).
| For applicative contexts we add \( [ ] \approx_p [ ] \).
| Note, we use the notation \( \overline{P} \) ambiguously for both the result of mapping \( P \) to \( \overline{P} \)
| and a term in \( \Lambda_c \) that is related to a term \( P \) in \( \Lambda_p \) via \( \approx_p \).

The relation \( \approx_p \) in Definition 4.2 is implicit. It is well-suited to capture the
different continuation representations from the above example, but it does not expose
the structure of the terms which stand for continuation points. A brief investigation
reveals that these terms are rather similar. If there is a proper term contained in the
continuation point-context, exactly one of the partitioning contexts covers it, and
therefore, each subterm appears exactly once in the representation. Furthermore,
empty contexts correspond to \( \lambda x.\overline{A}x \) and, putting the two observations together,
we see that the terms that are related to a continuation point are the same modulo
some occurrences of \( \lambda x.\overline{A}x \):

**Lemma 4.3.** Let \( \langle p, C[ ] \rangle \) be a continuation point. Furthermore, let \( P, V \in \Lambda_p \)
and \( \overline{P}, \overline{V} \in \Lambda_c \) such that \( P \approx_p \overline{P} \) and \( V \approx_p \overline{V} \). Define three term sequence
schema \( K_1^1, K_2^2, \) and \( K_3^3 \) for all representations of \( K_c \), i.e., \( \langle p, C[ ] \rangle \approx_p K_c \),
such that \( K_i^1 \equiv \lambda x.\overline{A}x, K_i^2 \equiv \lambda f.\overline{A}(K_i(f\overline{P})), K_i^3 \equiv \lambda v.\overline{A}(K_i(\overline{V}v)), \) and \( K_{i+1}^3 \equiv \lambda x.\overline{A}((\lambda x.\overline{A}x)(K_i x)) \). Then, a continuation point is related to exactly one of
the three schema, i.e., for all \( i \) and \( K_c \).
(i) \( \langle p, [ ] \rangle \approx_p K_i \),
(ii) \( \langle p, C[[ ]P] \rangle \approx_p K_i \), or
(iii) \( \langle p, C[V[ ]] \rangle \approx_p K_i \).

Furthermore, we can generalize this to
\[ \langle p, C[D[ ]] \rangle \approx_p [K_e D[ ]], \quad \text{iff} \quad \langle p, C[ ] \rangle \approx_p K_e. \]

**Proof.** First note that (i), (ii), and (iii) cover all possible cases of applicative contexts. One of them must match a particular applicative context. Furthermore, the proof of all three statements is naturally divided into two parts: one for \( i = 1 \) and one for \( i > 1 \). The latter is the same in all cases. For the former we demonstrate how to prove case (ii) as a typical example.

From the definition of \( \approx_p \) we know that for any context \( C[[ ]P] \) and finite number of contexts \( C_i[ ] \) which compose \( C[[ ]P] \), we have
\[ \langle p, C[[ ]P] \rangle \approx_p [\ldots [C_1[ ]], C_2[ ], \ldots C_n[ ]],. \]

For the base case we assume that \( C_n[ ] \neq [ ] \). Then, in (ii), \( C_n[ ] \equiv D[[ ]P] \) for some context \( D[ ] \) since \( P \) is the term next to the hole in the continuation-point context. This implies that
\[ \ldots [[C_1[ ]], C_2[ ]], \ldots D[[ ]P]], \equiv \lambda f. A([[\ldots [[C_1[ ]], C_2[ ]], \ldots D[ ]]], (fP)). \]

On the other hand, \( C[ ] \equiv C_1[ \ldots D[ ] \ldots ] \) and thus
\[ \langle p, C[ ] \rangle \approx_p [\ldots [[C_1[ ]], C_2[ ]], \ldots D[ ]]. \]

This proves the case for \( i = 1 \).

For the inductive case assume that the last \( i \geq 1 \) contexts in this sequence are empty, i.e. equal to [ ]. By factoring out the first one, we get
\[ \ldots [[C_1[ ]], C_2[ ]], \ldots [ ]], \equiv \lambda x. A([[\ldots [[C_1[ ]], C_2[ ]], \ldots ]], x)) \equiv \lambda x. A((\lambda x.Ax)(((\ldots [[C_1[ ]], C_2[ ]], \ldots ]), x)). \]

Thus we see that, as mentioned above, every empty context adds one term \( \lambda x.Ax \). Hence, \( \langle p, C[[ ]P] \rangle \approx_p K_{i+1} \) and this concludes the induction step.

The generalization follows immediately. \( \square \)

Lemma 4.3 indicates an important fact about the terms in the representation set of \( \langle p, C[ ] \rangle \): they are behaviorally indistinguishable in standard computation sequences with respect to \( \beta \)-steps. They invoke a continuation and, since continuations always remove the current context, none of the \( \lambda x.Ax \) ever plays a role in an evaluation.
Proposition 4.4. Define three term sets as in Lemma 4.3 with the initial terms 
\( \lambda x. A x \), \( \lambda f. A(K_c(f P)) \), and \( \lambda v. A(K_c(\forall v)) \). Then we can show that
\[
K_1, \ldots, \lambda x. A \ldots (i \times) \ldots x, \\
K_2, \ldots, \lambda f. A \ldots (i \times) \ldots (K_c(f P)), \\
K_3, \ldots, \lambda v. A \ldots (i \times) \ldots (K_c(\forall v)),
\]
are standard reduction sequences.

Proof. Clearly, \( K_i \equiv \lambda x. A M_i^x \) for some (open) term \( M_i^x \). Hence,
\[
K_{i+1} \longrightarrow_c \lambda x. A((\lambda x. A x)((\lambda x. A M_i^x)[x])) \\
\quad \longrightarrow_c \lambda x. A((\lambda x. A x)(A M_i^x[x := x])) \\
\quad \longrightarrow_c \lambda x. A(A M_i^x).
\]
and all are standard steps. But, the three \( M_i^x \)'s for the base cases are \( x \), \( (K_c(f P)) \), 
and \( (K_c(\forall v)) \), respectively. □

Proposition 4.4 says that all continuations related to a continuation point behave similarly when invoked; the difference is the number of abort operations. Thus, we can show that evaluations via the standard computation function and the C-rewriting system only differ in their outcome. First, we prove that the standard computation function mirrors C-transition steps as long as no continuation is invoked:

Lemma 4.5. Assume \( C[ ] \nearrow, C[ ] \), \( pP \nearrow, \) and \( U \nearrow, U \). The simulation of the rules (C1), (C2), and (C4) via \( \beta \Rightarrow \) respects \( \alpha \):

(i) if \( C[(\lambda x.P)U] \overset{C}{\longrightarrow} C[P[x := U]] \) then \( D[(\lambda x.P)U] \overset{\beta \Rightarrow}{\longrightarrow} D[P[x := U]] \) for any applicative context \( D[ ] \);

(ii) if \( C[AP] \overset{C}{\longrightarrow} P \) then \( C[AP] \overset{\beta \Rightarrow}{\longrightarrow} P \);

(iii) if \( C[CP] \overset{C}{\longrightarrow} P(p, C[ ] ) \) then \( C[CP] \overset{\beta \Rightarrow}{\longrightarrow} P[C[ ] ] \).

Proof. The first statement reiterates that \( \beta \)-steps are simulated independently of the context. Points (ii) and (iii) are consequences of Lemma 4.1. □

Things get more complicated when a continuation is invoked. The standard computation sequence contains a series of auxiliary moves in order to simulate the jump to a different context in the C-reduction sequence. Since proper simulation steps are interspersed in this detour, it is impossible to prove a corresponding lemma for (C3). However, a direct proof that continuation invocations are correctly implemented by the standard computation function is possible:
Lemma 4.6. Suppose \((p, C_0[ ] \equiv_p K_0, V \equiv_p V, \text{ and } U \equiv_p U)\). Then,
\[
C[(p, C_0[ ])V] \rightsquigarrow^+_C U \iff \overline{C}[K_0V] \rightsquigarrow_{sc}^+ U.
\]

Proof. The condition \(C[ ] \equiv_p \overline{C}[ ]\) is unnecessary for the antecedent since a continuation immediately performs some \(A\)-applications.

The equivalence is proved by an induction on the unique number of steps, \(n\), in the \(\rightsquigarrow^+_C\)-reduction sequence from \(C[(p, C_0[ ])]\) to \(U\). We proceed by case analysis on the structure of \(C_0[ ]\):

(apC1)\(C_0[ ] \equiv [ ]\): This case is trivial. It implies that
\[
K_0 \equiv K_1 \equiv \lambda x. Ax, \text{ or } \\
K_0 \equiv K_2 \equiv \lambda x. A((\lambda x. Ax)((\lambda x. Ax)x)), \text{ etc.}
\]

In any case, we have
\[
(p, C_0[ ])V \rightsquigarrow^+_C V \text{ and } K_0V \rightsquigarrow_{sc}^+ V.
\]

(apC2)\(C_0[ ] \equiv D[[ ]P]\) for some \(A_p\)-applicative context and term \(P\). Now we know from Lemma 4.3 that
\[
K_0 \equiv K_1 \equiv \lambda f. A(K_D(fP)), \text{ or } \\
K_0 \equiv K_2 \equiv \lambda x. A((\lambda x. Ax)(K_1 x)), \text{ etc.}
\]

where \((p, D[ ])V \equiv_p K_D\) and \(P \equiv_p P\). The two reduction sequences start out with
\[
C[(p, C_0[ ])V] \rightsquigarrow^+_C D[VP]
\]
and
\[
\overline{C}[K_0V] \rightsquigarrow_{sc}^+ K_D(VP).
\]

Next, we consider the possible evaluations of \(VP\) and \(VP\). The previous lemma reassures us that as long as the rule (C1) is used the context plays no role and, more importantly, the relation \(\equiv_p\) is preserved. The first transition step which does not conform to (C1) is the distinguishing criteria for the rest of the reduction sequence. Since this sequence is finite, four cases must be analyzed:
a) $VP \overset{C}{\longrightarrow}^+ (C_1) W$ where $W$ is a value. This means that $VP \overset{b}{\longrightarrow}_{sc}^+ W$ and we have the following development for the C-transition:

$$D[VP] \overset{C}{\longrightarrow}^+ D[W].$$

For the one according to $\overset{b}{\longrightarrow}_{sc}$ we get

$$K_D(VP) \overset{b}{\longrightarrow}_{sc}^+ K_D W.$$

By assumption we know that

$$D[W] \overset{C}{\longrightarrow}^m U \text{ with } m \leq n - 2.$$

From the definition of $\overset{C}{\longrightarrow}$ we see that,

$$D[W] \overset{C}{\longrightarrow}^m U \text{ iff } (p, D[]) W \overset{C}{\longrightarrow}^{m+1} U.$$

Thus, we can safely replace $D[W]$ by $(p, D[]) W$ since $m + 1 \leq n - 1$. But note, $(p, D[]) \overset{\alpha_p}{\longrightarrow} K_D$ and so, by inductive hypothesis, we get the desired conclusion.

b) $VP \overset{C}{\longrightarrow}^* (C_1) E[\tilde{A}Q]$ and $VP \overset{b}{\longrightarrow}_{sc}^* E[AQ]$ for some term $Q$ and applicative context $E[\ ]$. Comparing the two reduction sequences

$$D[VP] \overset{C}{\longrightarrow}^* D[E[\tilde{A}Q]] \overset{C}{\longrightarrow} Q$$

and

$$K_D(VP) \overset{b}{\longrightarrow}_{sc}^* K_D E[AQ] \overset{b}{\longrightarrow}_{sc}^* Q,$$

we see that both continue with related terms. From this point on, two developments are possible: the rest of the sequence either uses the (C3) rule or it doesn't:

b1) If $Q \overset{C}{\longrightarrow}^* U$ does not use (C3), then according to Lemma 4.5 $Q \overset{b}{\longrightarrow}_{sc}^* U$ is immediate.

b2) Suppose (C3) is used a first time. That means, that

$$Q \overset{C}{\longrightarrow}^* F[(p, F_0[]) W]$$

and also by Lemma 4.5 that

$$Q \overset{b}{\longrightarrow}_{sc}^* F[K_F W]$$
such that \((p, F_0[ ]) \approx_p K_F\). Since the reduction sequence is at least one step shorter, we can now apply our inductive hypothesis and this finishes case b).

c) \(VP \xrightarrow{C(K)} E[CQ]\) and \(VP \xrightarrow{sc} E[CQ]\). The reduction sequence according to \(\xrightarrow{sc}\) continues as:

\[K_D(VP) \xrightarrow{sc} K_D E[CQ] \xrightarrow{sc} \{K_D E[ ]\}\]

The transition rule (C2) accomplishes the capturing of this continuation in one step:

\[D[VP] \xrightarrow{C} D[E[CQ]] \xrightarrow{C} Q(p, D[E[ ]]).\]

By assumption \((p, D[ ]) \approx_p K_D\) and, hence, \((p, D[E[ ]]) \approx_p \{K_D E[ ]\}\) by Lemma 4.3. The rest of this subcase is as in b).

d) \(VP \xrightarrow{C(K)} E[(p, E_0[ ])W]\) and \(VP \xrightarrow{sc} E[K_E W]\) with \((p, E_0[ ]) \approx_p K_E\). This is an instance of the inductive hypothesis and the case (apC2) is finished.

(apC3) \(C_0[ ] \equiv D[P[ ]]\) for some \(\Lambda_\sigma\)-applicative context and value \(P\). Again, the respective reduction continuations are characterized by \(K_\lambda \equiv \lambda v. A(K_D(Pv)), etc. The two reduction sequences immediately arrive at the same constellation as in (apC2):

\[C[(p, C_0[ ])V] \xrightarrow{C} D[PV]\]

and

\[C[K_0 V] \xrightarrow{sc} K_D(PV).\]

The rest is analogous to the previous case. \(\Box\)

Putting the previous two lemmas together, the following theorem is obvious:

**Theorem 4.7 (Simulation).** For any program \(M \in \Lambda_c\), values \(V, V\) such that \(V \approx_p V\)

\[M \xrightarrow{C} V \iff M \xrightarrow{sc} V.\]

Since \(V \approx_p V\) implies \(V \equiv V\) for \(V \in \Lambda_c\), the theorem can be specialized:

**Corollary 4.8.** For any program \(M \in \Lambda_c\) whose result \(V\) is continuation-free,

\[M \xrightarrow{C} V \iff M \xrightarrow{sc} V.\]

A more general consequence is that \(eval_C\) is only defined if the program is equivalent to a value:
Corollary 4.9. For any program $M \in \Lambda_e$, there exists a value $V$ such that $\lambda_e \vdash M \overset{=} {\approx} V$ iff $\text{eval}_C(M)$ is defined.

Informally, these results mean that the C-machine is characterized by a standard computation function (and sequence) of a calculus modulo some syntactic difference. In order to eliminate this difference, we would have to change the standard reduction function in such a way that a term $K(C.M)$ evaluates to $MK$ for a continuation $K$. From the above definition of $[-]_e$, one can see that recognizing terms as continuations is possible. But one could easily construct such a term $K$ by hand and then the normal evaluation sequence would be preferable: without knowing the history of a term, it is impossible to know when to apply the new rule.

Although the difference cannot be eliminated, it is not stringent. The result of a batch computation is generally expected to be a basic constant and then, Corollary 4.9 assures us that we get the correct result back since we encode these values in $\Lambda$. Otherwise, if a non-basic value is the result, a sensible interpretation is impossible because these values represent machine behavior. On the other hand, if a machine is used interactively where intermediate results are saved, the user can only be interested in getting such values back for potential future use. In this case we are safe because of Proposition 4.4. All terms that are related to a continuation point are behaviorally equivalent. Thus, we can assume that $\text{eval}_C$ and the operational semantics of $\lambda_e$ are equivalent.

A disadvantage of the above theorem and corollaries is their dependence on the standard computation function of the calculus. One would prefer to interpret terms in a less operational way using the equivalence relation instead. Traditionally, one thinks of terms as functions from some set of basic constants to basic constants. A program is equivalent to a basic constant and hence, it is a nullary function. Following Morris [13] and Plotkin [14] we define two interpretations of terms. For all $n \geq 0$, the calculus interpretation of a term $M$ is the function

$$I^n_M = \{(N_1, \ldots, N_n, V) | \lambda_e \vdash MN_1 \ldots N_n \overset{=} {\approx} V\}$$

where the $N_i$ and $V$ are basic values. The machine interpretation of a term $M$ is the function

$$M^n_M = \{(N_1, \ldots, N_n, V) | \text{eval}_C(MN_1 \ldots N_n) \equiv V\}.$$  

---

5 For the following two theorems we must assume that constants are represented by an equivalence class of terms, the normal form being the typical representative. Alternatively, we could have introduced constants into $\Lambda_e$, but that would have complicated the treatment of fundamental properties.
Given these interpretations, the correspondence of the C-machine to the $\lambda_c$-calculus is independent of a standard computation function:

**Theorem 4.10.** For any program $M$ in $\Lambda_c$, its calculus and machine interpretation are the same for all $n \geq 0$:

$$I^n_M = M^n_M.$$  

**Proof.** The theorem is a consequence of the Church-Rosser Theorem, Corollary 4.8, and Corollary 4.9. □

Theorem 4.10 essentially says that the machine and the calculus interpret a program as the same function. Given that the classical $\lambda$-calculus is for reasoning about the equivalence of these functions, the question naturally arises what proofs in $\lambda_c$ mean.

Since the relation $\equiv_c$ is not a congruence relation, it is clear that $M \equiv_c N$ does not mean that $M^n_M = M^n_N$ for any $n > 0$. The relation $\equiv_c$ only compares programs that are already supplied with all their input arguments. Intuitively, the equivalence relation $\equiv_c$ equates the global control intentions of programs. The subrelation $=^c$ is more like $=_{\beta}$: it compares the functionality and local control structure of terms.

The question generalizes to what equality in $\lambda_c$ means for open expressions, i.e., whether equality is preserved under all possible interpretations [13]. Put differently, we are asking whether equality in the calculus implies operational equality. From the above discussion about $\equiv_c$ and $=^c$, we know that only $=^c$ implies operational equivalence. For $\equiv_c$ we need to make sure that the terms behave equivalently in all cases, then it also implies operational equivalence:

**Theorem 4.11.** For $M, N$ in $\Lambda_c$,

(i) if $\lambda_c \vdash M =_c N$, then $M \simeq_c N$, and

(ii) if $\lambda_c \vdash C[M] \equiv_c C[N]$ for all applicative contexts $C[ ]$, then $M \simeq_c N$.

**Proof.** The proof of (i) is easy. It is essentially a transcription of Plotkin's corresponding proof for the $\lambda_u$-calculus.

Part (ii) deserves some elaboration. Assume the hypothesis and without loss of generality assume that $M$ and $N$ are in $\Lambda_c$ proper. Let $D[ ]$ be a context such that $D[MN]$ is closed. Now, suppose that $eval_C(D[M])$ is defined and, furthermore, that it is a basic constant. By Theorem 3.10 and Corollary 4.9

$$\lambda_c \vdash D[M] \equiv_c eval_C(D[M]).$$
Depending on the role of the fill-in term during the evaluation, we have to
distinguish two cases. It is possible that the term in the hole is never a direct
component of a standard redex. Then it gets thrown away since the result is a basic
constant. The conclusion is immediate. Otherwise, at some point a closed form of
\( M \) or \( N \) is an immediate component of some redex in some applicative context. But
note, \( \lambda_c \vdash C[M] \cong_c C[N] \) implies \( \lambda_c \vdash C[M[x := L]] \cong_c C[N[x := L]] \) for all values
\( L \). Therefore, with the necessary generalization to multiple substitutions, we have

\[
\lambda_c \vdash eval_C(D[M]) \cong_c eval_C(C[M[\overline{\mathcal{E}} := \overline{L}]])) \cong_c eval_C(C[N[\overline{\mathcal{E}} := \overline{L}]]).
\]

Hence, by the Church-Rosser theorem, \( D[M] \) and \( D[N] \) produce the same result. \( \square \)

The inverse of both statements is false. This is inherited from the \( \lambda_\nu \)-calculus
for which Plotkin has already shown that it is consistent but not complete with
respect to \( \cong_C \).

The second point of Theorem 4.11 is important. Together with the above theo-
rems and corollaries it yields an interesting system for dealing with continuations.
The theorem implies that it is sufficient to consider all applicative contexts instead
of program contexts for behavioral considerations. Theorem 4.7 and Corollary 4.8
provide the basis for using the context rewriting rules in conjunction with the cal-
culus reductions; this is helpful when reasoning about redexes in arbitrary applicative
contexts. In the last section we briefly discuss some possible applications of this
to

5. Reasoning with the \( \lambda_c \)-calculus

In the preceding sections we have shown how the \( \lambda \)-calculus can be extended to a
control calculus. The resulting system is correct with respect to the \( C \)-rewriting
semantics which in turn is equivalent to a classical operational interpretation of a
denotational semantics. Together, the two characterizations form a syntactic theory
of control. The predominant use that we perceive is for symbolic manipulations of
programs, e.g., verification and evaluation.

With our first application, we return to the \( \Sigma^*_0 \)-function from subsection 2.3.
Recall that \( \Sigma^*_0 \) sums the numbers in a tree unless a \( 0 \) occurs in the tree, in which
case it returns a \( 0 \). Given a predicate \texttt{has-zero?} that tests whether a tree contains
a \( 0 \) or not, we can formulate and prove a correctness claim:

**Proposition 5.1.** For all number-trees \( t \), \( \Sigma^*_0 \) operationally satisfies its specification:

\[
(\Sigma^*_0 t) \cong_C (\text{if (has-zero? } t) \text{ then } 0 \text{ else } (\Sigma^*_0 t)).
\]
Proof. Let $C[ ]$ be an arbitrary applicative context and consider an application of $\Sigma^*_t$ in this context:

$$C[\Sigma^*_t t] \triangleq_c C[C \lambda k.k(Y_v E_k t)] \triangleq_c C[Y_v F t],$$

where

$$E_k \equiv \lambda s.\lambda t.\begin{cases} \text{if (mt? t) } & 0' \\ \text{if (zero? (num t)) (k')0'} \\ (+\text{(num t)} (+\text{(s(lon t)}) (\text{s(rson t)})])) \end{cases}$$

and

$$F \equiv E_k[k := (p, C[ ])].$$

At this point, we must show that the recursive function $Y_v F$ behaves correctly. To this end, we split the claim into two subcases:

(i) if $(\text{has-zero? t}) \equiv F$, then $D[Y_v F t] \triangleq_c D[\Sigma^*_t t];$

(ii) if $(\text{has-zero? t}) \equiv T$, then $D[Y_v F t] \triangleq_c D[(p, C[ ])'0'] \triangleq_c C[0'].$

For (i), observe that $(\text{zero? (num t)})$ can never be true and hence,

$$\begin{align*}
(\text{if (zero? (num t)) (k')0'}) (+\text{(num t)} (+\text{(s(lon t)}) (\text{s(rson t)})__))
\end{align*}
$$

$$=_c (+\text{(num t)} (+\text{(s(lon t)}) (\text{s(rson t)}))).$$

From this, (i) follows immediately.

The second claim we prove by an induction on the structure of a tree. Suppose that there is a 0 in the tree and that it is at the root. Then we have the following evaluation in any applicative context $D[ ]$:

$$D[Y_v F t] \triangleq_c D[(+\text{(p, C[ ])}'0']).$$

Otherwise, the evaluation takes the second if-branch and we have:

$$D[Y_v F t] \triangleq_c D[(+\text{(num t)} (+\text{(Y_v F (lon t)}) (\text{Y_v F (rson t)}))]).$$

Assuming that a 0 is in the left subtree, the inductive hypothesis yields

$$\ldots \triangleq_c D[(+\text{(num t)} (+\text{(p, C[ ])}'0')) (\text{Y_v F (rson t)})]$$

for the applicative context $D[(+\text{(num t)} (+[ \text{(Y_v F (rson t)})]))]$. But note, because of the abortive character of continuation representations in applicative contexts (see Proposition 4.4), this results in

$$\ldots \triangleq_c D[(p, C[ ])'0'].
For the final case, where the 0 is in the right subtree, the proof relies on part (i), but is otherwise similar to the above. □

This first application of the $\lambda_c$-calculus demonstrates how to reason about a specific use of $C$-applications. Since $C$- and $A$-applications abstract from an extension of continuation-passing style programming, a more general question is whether the $\lambda_c$-calculus can prove as many results as the $\lambda$-calculus about the respective $cps$'ed programs. The question naturally divides into two subproblems, namely, whether the $\lambda_c$-calculus-relations preserve equality in the $\lambda$-calculus, and vice versa. We refer to the first as the soundness and the second as completeness question.

As for soundness, the proof is a tedious but straightforward calculation. The soundness of the $\beta_c$-reduction is known from Plotkin's investigation of the $\lambda$-calculus [14]:

**Proposition 5.2 (Soundness).** For $M \in \Lambda_c$ and for $L \in \Lambda$, an abstraction,

If $M \equiv_c N$, then $[M]L =_\beta [N]L$ and,

if $M \not\equiv_c N$, then $[M]I =_\beta [N]I$.

**Proof.** The proof is a two-step procedure which for the most part can be carried out by a program. First, the left-hand side and the right-hand side of each rule is translated into the $\lambda$-calculus via $cps$. Then the resulting expressions are reduced until no $\beta$-redexes are left. For the $C_R$ and $A_R$ cases one needs the assumption that $L$ is an abstraction; in all other cases, the respective left-hand and right-hand terms are already equivalent. □

Unfortunately, the calculus is not complete:

**Proposition 5.2' (Incompleteness).** There are $M$ and $N$ such that for some abstraction $L$, $[M]L =_\beta [N]L$ but $M \not\equiv_c N$.

**Proof.** The proposition is a consequence of Theorem 3.8 and of Plotkin's value-calculus being a sub-calculus. An example is given by: $M \equiv (\omega\omega)y$ and $N \equiv (\lambda x.xy)(\omega\omega)$ where $\omega \equiv (\lambda x.xx)$. □

Thus far, we have used the $\lambda_c$-calculus as an equational extension of the C-rewriting system. With the exception of the induction step in Proposition 5.1 the C-reductions have played no role. The following proposition shows that they are indeed useful for local program transformations:

---

$^6$ See Plotkin's Translation theorem [14, p.146] which discusses the use of $cps$'ed programs for the simulation of a call-by-value abstraction in the traditional $\lambda$-calculus.
Proposition 5.3. Define $\lfloor \cdot \rfloor^k_c$ as a variant of $\lfloor \cdot \rfloor_c$:

\[
\begin{align*}
\lfloor [\ ] \rfloor^k_c & \equiv k \\
\lfloor [C[[ ]M]] \rfloor^k_c & \equiv \lambda f.A([C[ ]^k_c(fM)) \\
\lfloor [C[V[ ]] \rfloor^k_c & \equiv \lambda v.A([C[ ]^k_c(Vv)).
\end{align*}
\]

Then the following two statements hold for any applicable context $C[ ]$:

(i) $\lambda x.C[AM] \simeq_C \lambda x.AM$, and

(ii) $\lambda x.C[CM] \simeq_C \lambda x.C\lambda k.M[[C[ ]^k_c$.

Proof. (i) By Lemma 4.1 $\lambda_c \vdash \lambda x.C[AM] =_c \lambda x.AM$.

(ii) We prove the statement by an induction on the structure of the context $C[ ]$. If it is empty, we must show that $\lambda x.\mathcal{C}M \simeq_C \lambda x.\mathcal{C}\lambda k.Mk$. But this follows from $\mathcal{C}M \simeq_C \mathcal{C}\lambda k.Mk$, which obviously holds. Otherwise, assume without loss of generality that we have a context of the form $C[[ ]N]$. Then we get the conclusion by a simple calculation:

$\lambda x.\mathcal{C}((\mathcal{C}M)N) =_c \lambda x.\mathcal{C}[[\mathcal{C}\lambda k.M(\lambda f.A(\kappa(fN)))]]$

$\simeq_C \lambda x.\mathcal{C}\lambda k.(\mathcal{C}\lambda k.M(\lambda f.A(\kappa(fN))))[[C[ ]^k_c$

by inductive hypothesis

$=_c \lambda x.\mathcal{C}\lambda k.M[[C[ ]^k_c(fN))$

$= _c \lambda x.\mathcal{C}\lambda k.M[[C[ ]N]]^k_c$. □

The proposition is useful in two different areas: source-to-source transformations and compiler optimizations. Moving the $\mathcal{C}$-application to a procedure entrance can improve a programmer's understanding about the intension of a procedure body. At the same time it may save some cost since on various machine architectures it may be cheaper to label continuations at the invocation of a function.

For non-von Neumann machine architectures the reduction characterization of control operations is even more important. For example, a term rewriting machine [10] can reduce all (reduction) redexes in parallel. Only computation steps must be ordered, yet, this happens naturally. At first glance, it may appear that the reduction system exchanges short $\mathcal{C}$-rewriting steps for long chains of reductions, but these appearances are deceiving. Newly built $\beta_n$-redexes within right-hand sides of $\mathcal{C}_L$- and $\mathcal{C}_R$-steps can immediately be reduced. By the time a $\mathcal{C}$-application has reached the top of the term, it may already have finished the construction of
the continuation. The trade-offs between these two computational models clearly
deserve more investigation.

In essence the development of a syntactic theory of control has demonstrated
that our understanding of programming with control operations is far from com-
plete. It raises the question of what continuations really are. In denotational
semantics they are represented by functions. But this only works because the defi-
nitions are expressed in a particular style, namely continuation-passing style. Their
nature as programming objects remains concealed. We expect that a further in-
vestigation of the \( \lambda_c \)-calculus will deepen our understanding of programming with
continuations and the nature of control operations in programming languages.

**Acknowledgement.** We wish to thank Carolyn Talcott and Mitchell Wand for
their helpful discussions and comments on earlier drafts of this paper. Michael Dunn
helped to clarify the Soundness and Incompleteness Proposition. We also thank
John Gateley and Chris Haynes for their comments. The anonymous referees sug-
gested several improvements to the organization and presentation of the material.

Matthias Felleisen and Eugene Kohlbecker are IBM Research Graduate Fellows.
The material is partly based on work supported by the National Science Foundation
under grants DCR 85-01277 and DCR 85-03279.
References


