On Computational Interpretations of the Modal Logic S4
I. Cut Elimination

Jean Goubault-Larrecq
Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe
Jean.Goubault@panillac.inria.fr, Jean.Goubault@ira.uka.de

August 29, 1996

Abstract

A language of constructions for minimal logic is the \( \lambda \)-calculus, where cut-elimination is encoded as \( \beta \)-reduction. We examine corresponding languages for the minimal version of the modal logic S4, with notions of reduction that encodes cut-elimination for the corresponding sequent system. It turns out that a natural interpretation of the latter constructions is a \( \lambda \)-calculus extended by an idealized version of Lisp's `eval` and `quote` constructs.

In this first part, we analyze how cut-elimination works in the standard sequent system for minimal S4, and where problems arise. Bierman and de Paiva's proposal is a natural language of constructions for this logic, but their calculus lacks a few rules that are essential to eliminate all cuts. The \( \lambda_{S4} \)-calculus, namely Bierman and De Paiva's proposal extended with all needed rules, is confluent. There is a polynomial-time algorithm to compute principal typings of given terms, or answer that the given terms are not typable. The typed \( \lambda_{S4} \)-calculus terminates, and normal forms are exactly constructions for cut-free proofs. Finally, modulo some notion \( \approx \) of equivalence, there is a natural Curry-Howard style isomorphism between typed \( \lambda_{S4} \)-terms and natural deduction proofs in minimal S4.

However, the \( \lambda_{S4} \)-calculus has a non-operational flavor, in that the extra rules include explicit garbage collection, contraction and exchange rules. We shall propose another language of constructions to repair this in Part II.

1 General Introduction

This paper presents an answer to two dual questions. The first is: what language do we need to get a proofs-as-programs/\( \lambda \)-calculus-as-types correspondence with the modal logic S4? As already noted by Davies and Pfenning [DP95], the answer is a kind of \( \lambda \)-calculus augmented with polished versions of Lisp's `eval` and `quote` primitives. Our initial motivation for answering this question was intellectual curiosity: if intuitionistic and minimal logics have it [How80] and now classical logic too [Gri90] why not some modal logics? Moreover, the current interest in linear logic and the fact that the rules of sequent systems for linear logic look like those for S4 a lot, have recently stimulated some researchers [BDP95] into exploring functional interpretations for S4 in the hope of understanding linear logic better.

The second question is: what kind of type system can we impose on Lisp's `eval` and `quote` primitives that would make them usable in an ML-like strongly typed and polymorphic functional language? Or alternatively, how can we make type-safe versions of these primitives in a way that would be most expressive? We show that a good answer is given by augmenting ML's type system which is basically intuitionistic logic to the intuitionistic version of the logic S4.

---

*Research partially funded by the HCM grant 73127-06 from the European Union. This work started in July 1994 while I was at Bull, and was finished while I was at the University of Karlsruhe.

1 On leave from Bull Corporate Research Center, rue Jean Jaurès, F-78340 Les Clayes sous Bois.
The paper has four parts. Part I is a gentle introduction to the basic notions that we shall need later on: what eval and quote are, what the modal logic S4 is, both from the semantic and the proof-theoretic point of view; and what the notion of functional interpretation of proofs à la Curry and Howard is. We then analyze how cuts can be eliminated from sequent proofs, i.e., what the reduction rules for eval and quote should be from purely logical principles. We show that a slight extension of Bierman and De Paiva’s proposal [BP95] for the λ_S4-calculus is the right language to represent proofs and proof transformations. However, it is endowed with very non-operational reduction rules which make it a poor choice for a programming language. In fact, it does not explain much how evaluation and quoting works.

Part II presents another solution: the λevQ-calculus. We show how we are naturally led to this language which is much more complicated than λ_S4 but also much more operational. In fact, it includes an infinite tower of interpreters much as in Lisp encoded by λ-calculus. This language reveals how evaluation and quoting really work. Interestingly enough, such implementation details such as stacks arise naturally from purely logical principles. The λevQ-calculus can also be seen as an extension of the λ_S4-calculus in that there is a translation from the latter to the former that preserves convertibility and reducibility.

In Part III, we prove that the typed λevQ-calculus is confluent and that it is a conservative extension of the λ_S4-calculus: two typed λ_S4-terms are interconvertible if and only if their translations to λevQ-terms are as well. The same properties hold with both calculi extended by η-like rules. However, the untyped λevQ-calculus with η-like rules is not confluent: the status of the untyped λevQ-calculus without the η-like rules is unknown although we conjecture that it is confluent.

In Part IV, we examine the needed extensions to λ_S4 and λevQ that would make them suitable to interpret proofs in the classical version of S4, where double negation elimination is allowed. As is now well-known, integrating classical features allows one to represent control operators in the language of constructions, i.e., exception handling. Building on Parigot’s λµ-calculus, we define a classical version of λ_S4. However, there seems to be no hope of making it confluent. On the other hand, the classical λevQ-calculus is an extension of the λevQ-calculus with η-like rule based on Audebaud’s λµenv seems to have no such problem.

1.1 Plan of Part I

The plan of this Part I is as follows. In Section 2, we recall what Lisp’s eval and quote are, and argue that they provide useful functionalities. We then proceed to study S4 both as a logic and then as a possible type system for eval and quote. We give a short tour of the logic in Section 3. Then examine the problems involved when trying to eliminate cuts from proofs expressed in a sequent system for S4 in Section 4. Cut-elimination is by the Curry-Howard correspondence if the fundamental execution mechanism of the language. There are several pitfalls in doing this for S4 and we shall how these problems are solved in the λ_S4-calculus. We then recapitulate related works in Section 6.

2 Eval and Quote in Lisp

2.1 Short Description

In Lisp at least the very first implementations of programs and data structures were coded in memory in the same format by using so-called lists. Lisp lists are either the empty list nil a.k.a. () for cons-cells (x . y) where x and y are lists or symbols. The list (x1 . (x2 . (⋯(xn . nil ⋯))) is usually written (x1 x2 . . . xn). This universal data structure is not only used to represent ordinary data structures but also programs: when x1 evaluates to a function f this list is the program that applies the value of x1 to the values of x2 . . . xn. Besides, the special lists of the form (lambda (x1 x2 . . . xn) t) evaluate to the function that takes n arguments f and binds them to the symbols x1 . Γx2 . . . Γxn (which act as program variables) f and returns the value of the program t in this environment. Because data and programs share a common representation, Lisp is said to respect the programs = data equation.

This equation is somewhat exaggerated as it is rather an isomorphism than a genuine equality. This isomorphism is materialized by the pair of quasi-inverse functions eval and quote. The former takes a data structure and considers it as a program to execute (or equivalently, an expression to evaluate); that is, to execute (eval x) Lisp first executes x to yield a data structure, then eval views the latter as a program that
it executes again. On the other hand, the `quote` takes an expression (representing a program) and turns it into a data structure. I.e., a list which can then be handled by regular list operations: `(quote x)` is also abbreviated as `x` if `eval` leaves `x` unevaluated, and immediately boxes it as a Lisp data structure so that executing `x` yields `x` itself. `(eval 'x)` yields the result of the execution of `x`. (In short, `eval 'x` = `x`.)

Lisp also has a few other related constructs. The first is `kquote` which takes an argument `x` executes it, and quotes the result. (Therefore, we also have `(eval (kquote x)) = x`.) This is useful to build quoted data structures from other quoted data structures. Generalizing `quote` and `kquote` we have the backquote notation (called quasi-quote in Scheme) a modern variant of Lisp [CR91]): A backquote expression is an expression of the form `x` where `x` is a list possibly containing comma expressions of the form, `y`, where `y` are regular lists. (The comma is called `unquote` in Scheme.) Evaluating `x` then constructs the data structure `x` where all comma expressions, `y`, are first replaced by the result of evaluating `y`. This way, `x` is just the same as `x`, `x` is the same as `x` and, for example `(a , x)` is the same as `(list 'a x)` where `list` is the function that computes the list of values of its arguments. The function `kquote` is then operationally the same as `(lambda (x , (quote x)) x)` which turns an object into a quoted object.

### 2.2 What Good are Eval and Quote?

Let’s examine the uses of `eval` and `quote` in Lisp code.

A typical use was to simulate higher-order computations when Lisp did not have proper scoping rules yet (before Scheme [CR90] essentially). The correct rule for evaluating a λ-expression was done in Scheme and the languages of the ML family is to build a closure, i.e., a functional value that embodies both the code of the λ-expression and the current environment of bindings between variables and values. This closure can then be applied at a later time by reinstalling the stored environment and evaluating the body of the λ-expression in this environment.

In early Lisps like Lisp 1.5 [MCAE+62] this evaluation rule was not applied: the evaluation of `lambda` expressions was either illegal or just returned the `lambda` expression itself. For example, consider the `mapcar` function which applies its first argument `f` (a function) to all elements `x` of the list `l` in second argument `l` and returns the list of all `f(x)` in the same order. The only way to evaluate `(mapcar (lambda (x) (+ x 1)) l)` was to write `(mapcar (quote (lambda (x) (+ x 1))) l)` that is to pass not the function `f` but the piece of syntax that represented it. Then these Lisp systems were doing an implicit call to `eval` on data in functional position so the code inside `mapcar` automatically evaluated these pieces of syntax when applied to an argument.

This kind of dirty trick was one of the main uses of `eval` and `quote` in early Lisps. Besides, the mere presence of `eval` and `quote` as primitives enticed programmers into using this programming style; in turn, this produced code that was unnecessarily unreadable and impossible to debug. These reasons are certainly the main ones why the designers of the functional language Scheme [CR91] a cleaner version of Lisp I chose to leave these operations out. It was then recognized that not having them was no obstacle to programming and later languages like ML dispensed with them altogether.

However, there are some cases where a quoting mechanism (building a syntactic representation of a program) and an evaluation mechanism (executing the program given as syntax) are indispensable. The typical case is that of a network of machines where agents communicate with each other by sending out questions to other agents and interpreting the answers they get. These agents communicate through protocols (X Window, the Simple Network Management Protocol, `Sendmail` are three examples of protocols). Protocols are basically grammars for messages, i.e., they represent syntax of programming languages. These programming languages are usually thought of as being much simpler than general-purpose programming languages like Lisp or ML but it isn’t necessary so. The `Sendmail` protocol in particular is a language based on rewrite rules and can simulate any Turing machine.

Communicating through a protocol means the following. The sender of a message first builds the message using a few primitives (library functions) that build up messages from basic information and then sends it. The receiver decodes the message by doing a case analysis on all possible ways in which the message was built and loops: the receiver is an interpreter for the message it has received. While building a message is quite similar to building a program in Lisp via backquote-expressions decoding the message is analogous to evaluating it. The only difference with `eval` and `quote` is that the language of messages (the protocol) does not need to coincide with the programming language each party is programmed in.
Protocol interpreters are not hard to write usually but take time. Moreover protocols are usually specialized and in particular less expressive or less convenient than a general-purpose language like Lisp or ML. A simple solution would then be to use Lisp itself as a protocol: building messages would be a matter of using backquote-expressions interpreting them would mean applying `eval` on them. This may be overgeneral in some cases but it solves the problem of extending the protocol trivially (just add new definitions to the Lisp interpreter) it is easy to implement (just call `eval`) and in general provides some added comfort in distributed applications.

For example in the Magic Cap operating system (General Magic Inc.) agents located on networked appliances or computers are programmed in a special language named Telescript and communicate by sending Telescript scripts (programs) to each other. Although Telescript is not Lisp messages really are Telescript programs in some byte-code format and are decoded by executing them on the receiving appliance. This is a typical use of non-Lisp variants of `eval` and `quote`. (See [Rei94] for a description of Telescript TIs planned uses and other projects in the same direction.)

To take another example a variant of Lisp called Wool was chosen as implementation language for a network management project at Bull amongst other well-known languages like C++ because commands could be issued to remote machines by just sending them a piece of Lisp code that can then be interpreted remotely by `eval` on the local agent running Wool. As the format of Lisp code does not change (a list printed as nested expressions delimited by parentheses and then transmitted as a sequence of characters) these codes are always recognized by the receiving machine whatever the version of the code.

The only problem is that when we send a piece of code to call some function named \( f \) in the sending machine's environment we have no guarantee that the receiving machine will have a function named \( f \) as well and that it will execute as we expect it to. In general security is a concern. Lisp was not designed to enforce security in programming but Telescript allows receiving machines to protect themselves against ill-behaving incoming scripts by setting a few limits on usable resources (cpu ticks, memory usage) and so on and the same can be done with Lisp (or Scheme [MIT95]). ML was designed to enforce some degree of security through the use of a reasonably expressive type system; a type is a formula for a specification and a program of this type obeys the specification. There are some security holes in the ML type system — for example recursion is allowed so termination is not guaranteed — but it is balanced by the decidability of type-checking. Trying to design a type system for `eval` and `quote` is in our opinion a laudable goal. We can see this work as providing a logical basis for this endeavour.

3 The Logic S4

The logic notations we use are standard: \( \land \) is conjunction, \( \lor \) is disjunction, \( \Rightarrow \) is implication. \( \bot \) is a special constant denoting false; \( \top \) is another constant. \( \exists \) is existential quantification. \( \forall \) is universal quantification. \( \land \) is a conjunction. \( \lor \) is a disjunction. \( \Rightarrow \) is an implication. \( \bot \) is the bottom element. \( \top \) is the top element. \( \exists \) is an existential quantifier. \( \forall \) is a universal quantifier.

We shall deal with various languages of terms. A term language is given by a set of operators with specified arities (nullary operators are called constants) and a set of binding constructs like \( \lambda \). The terms \( u, v, \ldots \) in the language are the constants \( \theta \), the applications \( f(u_1, \ldots, u_m) \) of \( m \)-ary operators \( f \) to a list of \( m \) terms \( u_1, \ldots, u_m \), for binding terms like \( \lambda x : u \) which bind the variable \( x \) in the term \( u \). A substitution \( \sigma \) is any map from variables to terms. Applying a substitution \( \sigma \) to a term \( u \) yields a new term \( u \sigma \) where the substitution process renames bound variables to avoid variable capture [Bar84]. The composition \( \sigma \sigma' \) of two substitutions \( \sigma \) and \( \sigma' \) is defined as the unique substitution such that \( (u \sigma) \sigma' = u(\sigma \sigma') \) for every term \( u \).

We shall often use rewrite systems. A rewrite system [DJ90] is a set \( R \) of rewrite rules of the form \( l \Rightarrow r \) where \( l \) is a term called the left-hand side and \( r \) is a term called the right-hand side with term meta-variables \( u, v, \ldots \) that are to be instantiated by arbitrary terms; such a rule is actually a scheme to denote all its instances \( l \sigma \Rightarrow r \sigma \). We write \( u \Rightarrow v \) or \( u \Rightarrow^* v \) when we wish to make clear which rewrite system we use to say that \( u \) contracts for \( v \) in \( R \). This means that \( v \) is obtained from \( u \) by replacing a given occurrence of \( l \) by \( r \) where \( l \Rightarrow r \) is an instance of a rule in \( R \). We write \( u \Rightarrow^* v \) or \( u \Rightarrow^* v \) to say that \( u \) rewrites in zero or more steps to \( r \) and \( u \Rightarrow^* v \) or \( u \Rightarrow^* v \) to say that \( u \) rewrites in one or more steps to \( v \).
3.1 A Hilbert-Style System

We are interested in the modal logic S\(_4\). This is a propositional logic with additional \(\square\) and \(\Diamond\) operators: if \(\Phi\) is a formula then \(\square \Phi\) means “necessarily \(\Phi\)” for “in all possible futures \(\Phi\)” and \(\Diamond \Phi\) means “possibly \(\Phi\)” for “there is a future at which \(\Phi\)”. The interpretation in term of possible futures is what makes S\(_4\) more specifically a temporal logic. The deduction rules of S\(_4\) are usually presented as a Hilbert system:

- Rules:
  - \((\text{MP})\) from \(\Phi \land \Phi \rightarrow \Psi \) infer \(\Psi\);
  - \((\text{Nec})\) from \(\Phi \) infer \(\Box \Phi\).

- Axioms:
  - All propositional tautologies; if we choose as basic set of connectives for propositional logic the set \{\(\Rightarrow, \bot\}\) then it is enough to have:
    - \((s)\) \(\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3 \Rightarrow (\Phi_1 \Rightarrow \Phi_2) \Rightarrow (\Phi_1 \Rightarrow \Phi_3)\);
    - \((k)\) \(\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3\);
    - \((c)\) \(\square (\Phi \Rightarrow \bot) \Rightarrow \bot \Rightarrow \Phi\);
  - The modal axioms for S\(_4\):
    - \((K)\) \(\Box(\Phi_1 \Rightarrow \Phi_2) \Rightarrow \Box \Phi_1 \Rightarrow \Box \Phi_2\);
    - \((T)\) \(\Box \Phi \Rightarrow \Phi\);
    - \((4)\) \(\Box \Phi \Rightarrow \Box \Box \Phi\).

where \(\Box \Phi \Gamma \Phi_1 \Gamma \Phi_2 \Gamma \Phi_3\) range over all formulas. We can define the other connectives as abbreviations: \(\neg \Phi \equiv \Phi \Rightarrow \bot\); \(\vee \Phi_1 \vee \Phi_2 \equiv \neg \neg \Phi_1 \Rightarrow \neg \neg \Phi_2 \Rightarrow \bot \Gamma \text{ and } \Diamond \Phi \equiv \neg \neg \neg \Phi\).

This is classical S\(_4\). We could without doubt provide a functional interpretation for classical S\(_4\) but the latter includes classical propositional logic which has non-obvious functional interpretations by itself. We shall therefore restrict ourselves to intuitionistic versions of S\(_4\). To achieve this goal we don’t consider axiom \((c)\) above and get minimal S\(_4\). We cannot define the other logical connectives in this system \(\Gamma \sigma\) we have to axiomatize them. In this paper we won’t care about \(\vee\) or \(\Diamond\) (which is as hard to interpret functionally as the existential quantifier in first-order logic) and we won’t care about \(\land\) either. This leads us to not only an intuitionistic but in fact a minimal version of S\(_4\).

This particular system of intuitionistic S\(_4\) with only connective \(\Rightarrow\) will simply be called S\(_4\) in the sequel. Should we wish to the original logic we’d talk about classical S\(_4\).

3.2 A Functional Interpretation for S\(_4\)

As in ordinary Hilbert-style presentations of intuitionistic logic the axioms \((s)\) and \((k)\) are interpreted as the types of the two combinators \(S\) and \(K\) respectively and rule \((\text{MP})\) is interpreted as typing the application of one term to another. Then we can view the \(SK\) calculus as being implemented through the following reduction rules:

- \(Suw \rightarrow uw(uw)\);
- \(Kuw \rightarrow uw\);

where \(uw\) are terms \(\Gamma\) application is denoted by juxtaposition (i.e., \(\Gamma uv\) means \(u\) applied to \(v\)) and application is right-associative (so \(Suw\) is really \((Su)v\) \(u\Gamma uv(sv)\) is really \((uw)(sv)\) and \(Kuw\) is really \((Ku)v\)).

Typed terms are seen as constructions for a particular proof \(\Gamma \text{ in short } \Gamma \text{ term } \Rightarrow \text{ proof } \Gamma \text{ and the conclusion of the proof is the type of the term (in short } \Gamma \text{ type } \Rightarrow \text{ proposition } \). This is the spirit of the Curry-Howard functional interpretation for combinatory logic.

These reduction rules preserve the types i.e. theorems proved by a proof represented by \(Suw\) are also proved by the simpler proof \(uw(uw)\). However it is not clear in which way reducing a term produces a simpler term. It will be clearer on sequent systems (see Section 3.4).
It remains to interpret axioms (K) \( \Gamma(T) \Gamma(4) \) and rule (Nec). The basic idea of this work can be explained as follows. If \( \Phi \) is a type (a proposition) then \( \Box \Phi \) is a type of "boxes" containing an object of type \( \Phi \).

Axiom (K) now can be seen as an application of (MP) inside boxes: if \( u \) has type \( \Box (\Phi_1 \Rightarrow \Phi_2) \) \( (u \) is a box containing a function of type \( \Phi_1 \Rightarrow \Phi_2 ) \text{ and } \text{r has type } \Box \Phi_1 \) \((r \) is a box containing an argument of type \( \Phi_1 ) \Gamma \) then some term \( \Box \Phi_2 \) (where \( \text{r} \) has axiom (K) as type \( \Gamma \text{ and is written infix for convenience} \) has type \( \Box \Phi_2 \).

That is \( u \times r \) is a box that contains the application of \( u \times r \) to \( r \).

On the other hand, axiom \( (T) \) is interpreted as the type of an "unbox" operation \( \Box \text{ and rule (Nec) as the typing rule for the converse operation \( \text{.} \) We argue that \( \Box \) bears strong similarities to Lisp's \texttt{eval} primitive \( \text{and } \Box \text{ looks a lot like Lisp's \texttt{quote} special form. Indeed } \text{given a Lisp expression } u \texttt{('quote } u \texttt{')} \) (often abbreviated \( \texttt{'u} \) is a (boxed) data structure that \( \text{when we apply eval on it yields the value of } u \).

This data structure is built by list building \( \text{operations on } \Gamma \text{ based on the cons couple-building primitive } \text{analogous to our } \times \text{ operation. (But not quite the same as Lisp does not represent applications } uv \text{ as couples } (u,v) \text{ but as } (u,(v,nil))); \text{ this is inessential.} \) This data structure can also contain boxed representations for \( \lambda \text{-expressions } \Gamma \text{ which we can recreate using combinators: for example the Lisp expression } \texttt{(('lambda } (x) x)} \text{ becomes } \texttt{'(lambda } (x) x) \text{'; we could also build it by hand as } \texttt{S'*K*K'Gamma} \text{ and this would be similar to building the Lisp list } \texttt{(lambda } (x) x) \text{ explicitly by executing the expression } \texttt{(list 'lambda } (list 'x) 'x). \) This suggests the following reduction rules:

- \( \Box (u) \rightarrow u; \)
- \( \Box (u \times v) \rightarrow (u)(v); \)
- \( \Box (uv) \rightarrow (u) \times (v); \)

which indeed preserve the types \( \text{ie. are valid proof transformation rules.} \)

There is a dual interpretation of \( \times \) and \( \Box \). For example \( \Box (u)(v) \) reduces to \( uv \text{ but Lisp's } \texttt{('eval } u \texttt{) (eval } v \texttt{)} \) does not reduce to anything. Another way of interpreting \( \times \) and \( \Box \) in this case is by means of the backquote notation \( \texttt{'} \) (Scheme's \texttt{quasiquote} special form \[\text{CR91}\]) and the comma notation \( \text{,} \) (Scheme's \texttt{unquote} special form). We can interpret \( \Box (u)(v) \) as the analogue of \( (u,v) \) which is precisely equivalent to \( (u,v) \).

Then \( \Box \) would be Lisp's \texttt{'}, or Scheme's \texttt{quasiquote} and \( \Box \) would be both \texttt{eval} and \texttt{unquote}. This justifies the following reduction rule which again preserves the types:

- \( \Box (u) \rightarrow u; \)

and which is the equivalent of the Lisp reduction \( u \rightarrow u. \) The fact that we confuse \texttt{eval} and \texttt{unquote} cannot be considered a departure from the spirit of \texttt{eval} and \texttt{quote}. However its kind of rule won't be needed to interpret cut-elimination; at best we can interpret this as an attempt to an \( \eta \)-like rule in the calculus. (Another would be \( \Box (uv) \rightarrow (u) \times (v); \).

The only rule that remains to interpret is \( (4) \). The idea is that one major difference between our language and Lisp is that Lisp is not statically typed; in particular \( \Gamma \text{ given any Lisp object } \Gamma \text{ we can quote it on the fly. The operation that does this is called } \texttt{kwote} \text{ in Lisp } \Gamma \text{ and would have type } \Phi \Rightarrow \Box \Phi. \)

Adding the latter as an axiom would not only entail axiom \( (4) \text{ but it would essentially amount to reduce the logic to pure intuitionistic logic } (\Box \Phi \text{ being always identical to } \Phi \text{ we'd not need } \Box \text{ at all). As } \Phi \Rightarrow \Box \Phi \text{ is in general not provable in } \text{S4} \text{ we cannot quote any arbitrary object. In particular the function that maps the variable } x \text{ to } \Box x \text{ has no meaning; compare the problematic Lisp expression } \texttt{(lambda } (x) \texttt{(quote } x) \texttt{)} \text{ where } x \text{ inside the } \texttt{quote} \text{ expression is understood as } \textit{not} \text{ in the scope of the header of the } \texttt{lambda} \text{-expression.} \)

Axiom \( (4) \) for the other hand expresses a way of relaxing the impossibility of quoting arbitrary objects: we can quote quoted objects. The property of being quotable (at run-time) is a characteristic of the type of the object: let's say that a type \( \Phi \) is \texttt{boxable} if and only if \( \Phi \Rightarrow \Box \Phi \text{ is provable } \text{i.e. if there is a term } \texttt{kwote}_q \text{ of type } \Phi \Rightarrow \Box \Phi. \) Axiom \( (4) \) says that all boxed types are boxable. There are many more boxable types: if we add universal quantifications to \( \text{S4} \text{ i.e. if we consider second-order propositional intuitionistic S4} \text{ so as to be able to speak of data types } \text{then } \text{Booleans} \text{, } \text{Integers} \text{, } \text{products and sums of boxable types are still boxable. But function types are in general not boxable: for example } \texttt{A} \Rightarrow \texttt{B} \text{ where } A \text{ and } B \text{ are propositional variables is not boxable.} \)

Our interpretation of this phenomenon is the following. In Lisp \texttt{eval} and \texttt{quote} mainly have one use that is not subsumed by the use of higher-order functions and closures. This use is \textit{reification} i.e. producing data
structures representing arbitrary programs which can then be transmitted over through a network or saved on disk and then be evaluated by a remote machine or reloaded from disk and evaluated. This can be done in a portable manner on usual data structures like Booleans, integers, products, sums but not on functions in general: there is no (simple at least) way of transmitting a possibly compiled function over to a remote machine of a possibly totally different architecture so that the function works as expected on the remote machine. The restriction on boxable types imposed by S4 seems to stick well to this intuition although the reason why S4 looks like the right system for enforcing safety of Lisp's `eval` and `quote` completely eludes us at the moment.

### 3.3 Semantics

The most well-known semantics of modal logics is the so-called Kripke semantics. It is based on the notion of Kripke frames which are triples $< W, R, \mathcal{I}>$ where $W$ is a non-empty set of so-called possible worlds $w$, $R$ is a binary relation between worlds called the accessibility relation (alternatively $\Gamma(W, R)$) and $\mathcal{I}$ is an interpretation that is a function mapping propositional variables to the set of worlds where they hold. In the case of S4 it is enough to consider reflexive and transitive frames i.e. frames where $R$ is a preorder $\geq$. (This preorder is the “future of” relation $\mathcal{I}$ the temporal interpretation.)

The semantics of classical S4 is then given by the relation $w \models \Phi$ ($\Phi$ holds in world $w$) defined as follows:

- $w \models A$ if and only if $w \in [A]$;
- $w \models \Phi_1 \Rightarrow \Phi_2$ if and only if $w \models \Phi_1$ implies $w \models \Phi_2$;
- $w \models \Box \Phi$ if and only if for every world $w' \geq w, \Gamma w' \models \Phi$.

We can therefore guess that a semantics for intuitionistic S4 would be similar except that all conditions above are to be considered as intuitionistically provable. In particular $w \models \Phi_1 \Rightarrow \Phi_2$ if and only if $w \models \Phi_1$ intuitionistically implies $w \models \Phi_2$. This however needs to be proved. We leave this to the final report.

### 3.4 A Sequent System

There are few interesting proof-theoretic results on Hilbert-style systems and it is more fruitful to consider natural deduction systems or sequent systems to this end. It is more customary to deal with natural deduction proof systems and to translate these proofs as $\lambda$-terms. However the basic execution mechanism which is the elimination of detours in natural deduction proofs appears more clearly on Gentzen-style sequent systems where it corresponds to cut-elimination. Since we are looking for such an execution mechanism we shall first consider a sequent system for S4.

We claim that the sequent system LS4 shown in Figure 1 is a sequent system for S4 where a boxed context $\Gamma$ is a set of boxed formulas $\Phi$ of formulas of the form $\Box \Phi$. The restriction that $\Gamma$ be a boxed context in rule $\Box R$ is essential. The proof that this system really defines S4 is given in Appendix A.

We have already decorated sequents with terms $\Gamma \vdash u : \Phi$ as to reflect the functional interpretation of Section 3.2. As is customary the sequent (a typing judgment) $\Gamma \vdash u : \Phi$ consists of a context $\Gamma$ which is now a map from variable names $x \Gamma y \Gamma \ldots$ to formulas (types) $\Phi$ and the right hand-side declares that $\Phi$ follows from the assumptions $\Gamma$ and that the constructive content of the proof leading to this sequent is $u$ (alternatively the term $u$ is of type $\Phi$). The language of constructions (of terms) is the $\lambda$-calculus [Bar84] augmented with an $\otimes$ operator and more; in particular we shall need additional operators to define a reasonable construction $u'$ from a term $u$ but we refrain for now from simply using an $\otimes$ operator and defining $u'$ as $u$. The reason of this and what the notation $u'$ really means will be elucidated in Section 4.1.

We write $u[v/x]$ the result of substituting $v$ for the free occurrences of $x$ in the term $u$ if the term $u$ where free occurrence assumes its usual meaning in the $\lambda$-calculus. ($\otimes$ being the only variable-binding operator.) To avoid problems we adopt Barendregt's variable naming convention [Bar84] that no bound variable occurs free in any $\lambda$-term.

On the other hand if contexts are maps from variables to types then $\Gamma, \Gamma'$ cannot mean set union: we define it as the overriding of $\Gamma$ by $\Gamma'$ i.e. the context that maps $x$ to its type in $\Gamma'$ if it exists and otherwise
to its type in \( \Gamma \) if it exists. Doing so poses no problem whatsoever for all rules but \((Cut)\). Hence we also adopt the following convention:

**Definition 3.1 (Naming Convention)** Given a cut between two sequents \( \Gamma_1 \vdash u_1 : \Phi_1 \) and \( \Gamma_2 \vdash u_2 : \Phi_2 \), we assume that no variable \( x \) occurs both in \( \Gamma_1 \) and in \( \Gamma_2 \).

Otherwise we rename free variables in one of the sequent (and the proofs above). Observe that if we consider sequents as binding all variables on their left-hand sides this is a variant of \( \alpha \)-renaming. To tell it with the vocabulary of linear logic [GLT89] we formulate all non-\((Cut)\) rules as additive rules but we are forced to formulate \((Cut)\) in a multiplicative fashion. This is essential to avoid problems in formulating cut-elimination strategies.

We also make the following important remark. The *weakening rule*:

\[
(Weaken) \quad \Gamma \vdash u : \Phi \\
\Gamma', \Gamma \vdash u : \Phi
\]

where \( \Gamma' \) is any additional set of typing assumptions \( x_i : \Phi_i, 1 \leq i \leq n \). That is whenever we have a proof \((\pi)\) of \( \Gamma \vdash u : \Phi \) there is another proof \( \Gamma', (\pi) \) of \( \Gamma', \Gamma \vdash u : \Phi \). Just add \( \Gamma' \) to the left of all sequents in \( (\pi) \) from the bottom up until we reach axioms \((Ax)\) or instances of \((\Box R)\). (We cannot go over to the premises of \((\Box R)\).)

### 4 Deriving a Language of Constructions

The sequent system of Section 3.4 includes the \((Cut)\) rule. We wish to eliminate it for at least two distinct reasons.

The first one is related to automated deduction: if we don’t need \((Cut)\) then the only thing to do to find a proof of a given proposition is to try and construct a proof from the bottom up (from the proposition to axioms \((Ax)\)). In doing this we are guided by the very structure of the formula to prove. Notice indeed that the sequent system is designed so that having chosen one formula from the goal sequent there is at most one rule that applies to get to a new sub-goal. It is a well-known fact that in the classical S4 case that the deduction system of Section 3.4 without \((Cut)\) is sound and complete [Gor93]. We shall prove it syntactically in the intuitionistic case.

The second reason why we want to eliminate cuts is that is to apply cut-elimination: not just to wish cuts were not there but stems from the functional interpretation. In this interpretation proofs (with or without \((Cut)\)) are terms in a programming language and cut-elimination is the very operational semantics of the language. We shall therefore derive rewrite rules by examining how cut-elimination works.
4.1 Cut Elimination

The usual way of eliminating cuts is the following: given an arbitrary proof in a Gentzen system we permute rules so that instances of (Cut) bubble up the proof. When we reach instances of (Ax) a cut with such a rule can be seen as yet another instance of (Ax) and the cut disappears. The only remaining difficulty is to show that this process of bubbling up cuts terminates.

Bubbling up cuts in general changes the proofs hence the constructions — or terms — that label the formulas on the right-hand side of sequents. This corresponds to reduction rules of the term language we now demonstrate. In any of the rules let’s call active formula the formula that the rule builds either on the left or on the right: \( \Phi \) in \((Ax)\Gamma \Phi \Rightarrow \Phi_2 \) in \((\Rightarrow L)\) and \((\Rightarrow R)\Gamma \Phi \Rightarrow \Phi_2 \) in \((\Box L)\) and \((\Box R)\).

Given two sub-proofs \((\pi_1)\) and \((\pi_2)\) performing a cut on their end sequents yields a proof of the form:

\[
\begin{array}{c}
\vdots \\
\Gamma_1 \vdash u_1 : \Psi_1 \\
\vdots \\
\Gamma_2, x_1 : \Psi_1 \vdash u_2 : \Psi_2 \\
\hline
\hline
(Cut)
\Gamma_1, \Gamma_2 \vdash u_2[x_1/x_1] : \Psi_2
\end{array}
\]

There are several cases to permute (Cut) with the rules above it. All of them are well-known except possibly the last one which deals with the \((\Box L)\) and \((\Box R)\) rules.

Before we embark on describing how this is done recall that weakenings are admissible: if \((\pi)\) is a proof of \(\Gamma \vdash u : \Phi\) then we write \(\Gamma', (\pi)\) its weakening by \(\Gamma'\).

We now proceed to permuting cuts up \(\Gamma\) and there are two cases.

4.1.1 Case 1: \(\Psi_1\) is active on both sides

In the first case \(\Gamma \Psi_1\) is active both on the right-hand side of \(\Gamma \vdash u_1 : \Psi_1\) and on the left-hand side of \(\Gamma, x_1 : \Psi_1 \vdash u_2 : \Psi_2\). We have the following cases of which the first one is degenerate:

\((Ax)\) \(\Gamma = \Gamma, x : \Phi \Gamma u_1 = x\) and \(\Psi_1 = \Phi\); we transform:

\[
\begin{array}{c}
\vdots \\
(\pi_2)
\end{array}
\]

\[
\begin{array}{c}
(Ax) \\
\hline
\Gamma, x : \Phi \vdash x : \Phi \\
\vdots \\
\Gamma_2, x_1 : \Phi \vdash u_2 : \Psi_2 \\
\hline
\hline
(Cut) \\
\Gamma, x : \Phi, \Gamma_2 \vdash u_2[x/x_1] : \Psi_2
\end{array}
\]

into:

\[
\begin{array}{c}
\vdots \\
\Gamma, (\pi_2)[x/x_1]
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x : \Phi, \Gamma_2 \vdash u_2[x/x_1] : \Psi_2
\end{array}
\]

where \(\Gamma, (\pi_2)[x/x_1]\) is the result of replacing \(x_1\) by \(x\) throughout \((\pi_2)\) — this effectively contracts two assumptions for \(\Phi\) into one — and then weakening \((\pi_2)[x/x_1]\) by \(\Gamma\).

Computationally this reduction has no effect as the end construction is \(u_2[x/x_1]\) in both cases.

\((\Rightarrow R)/(\Rightarrow L)\ u_1 = \lambda x : u \Gamma \Psi_1 = \Phi_1 \Rightarrow \Phi_2 \Gamma x_1 = y \Gamma u_2 = v[y/w/z]\) \(\Gamma \Psi_2 = \Phi_3 \Gamma\) and we transform:

\[
\begin{array}{c}
\vdots \\
(\pi_1)
\end{array}
\]

\[
\begin{array}{c}
(\pi_2)
\end{array}
\]

\[
\begin{array}{c}
(\pi_3)
\end{array}
\]

\[
\begin{array}{c}
(\Rightarrow R) \\
\hline
\Gamma_1, x : \Phi_1 \vdash u : \Phi_2 \\
\hline
\hline
(Cut) \\
\Gamma_1, \Gamma_2 \vdash v[y/w/z][\lambda x - u/y] : \Phi_3
\end{array}
\]

\[
\begin{array}{c}
(\Rightarrow L) \\
\hline
\Gamma_2, y : \Phi_1 \vdash v[y/w/z] : \Phi_3
\end{array}
\]

\[
\begin{array}{c}
(\Rightarrow R) \\
\Gamma_1 \vdash \lambda x : u : \Phi_1 \Rightarrow \Phi_2 \\
\end{array}
\]
Because of the variable naming convention $\Gamma([u/v][z]/[\lambda x \cdot u]/[y])$ is in fact $v[(\lambda x \cdot u)[u/v][z]/[v]][w/x]$ and $v[u/v][z]/[w/x]$ is $v[u/v][z]/[\lambda x \cdot u]$ in detail shortly.) So cut-elimination here means transforming $v[(\lambda x \cdot u)[u/v][z]/[v]][w/x]$ into $v[u/v][z]/[w/x]$ and this is precisely the $\beta$-reduction rule.

$$(\Box R)/(\Box L) \quad \Gamma_1 \Gamma' \Gamma_2 \Gamma_3$$ where $\Gamma$ is boxed $\Gamma \Gamma_1 = v^1[\Psi_1 = \Box \Phi \Gamma_{1} \Gamma_{1} = y \Gamma_{1} \Gamma_{1} = u[y/x] \Gamma_{1} \Gamma_{2} = \Phi_{2} \Gamma_{2}$ and we transform:

$$\begin{array}{ll}
\Gamma_1 \vdash v^1 : \Phi_1 & \Gamma_2 \vdash u^1 : \Phi_2 \\
\vdots & \vdots \\
\Gamma \vdash v^1 : \Phi & \Gamma_2 \vdash u^1 : \Phi_2 \\
\hline
\Gamma, \Gamma' \vdash v^1 : \Phi & \Gamma_2 \vdash u^1 : \Phi_2 \\
\hline
\end{array}$$

Into:

$$\begin{array}{ll}
\Gamma, \Gamma' \vdash v^1 \Phi & \Gamma_2 \vdash u^1 : \Phi_2 \\
\vdots & \vdots \\
\Gamma \vdash v^1 : \Phi & \Gamma_2 \vdash u^1 : \Phi_2 \\
\hline
\Gamma, \Gamma' \vdash v^1 \Phi & \Gamma_2 \vdash u^1 : \Phi_2 \\
\hline
\end{array}$$

Here the cut bubbles up more easily than in the previous cases. Its computational interpretation is that $u[y/x]$ rewrites to $u[y/x]$ that $v^1$ rewrites to $v$. This suggest that we define $v^1$ as $v$ and use the reduction $(v) \rightarrow v$. But we won’t be able to do this because reasons we explain at the end of this section.

4.1.2 Case 2: $\Psi_1$ is inactive on some side

In the second case $\Psi_1$ is not active on the right-hand side of $\Gamma_1 \vdash u^1 : \Psi_1$ (then $(\pi_1)$ ends with a rule acting on the left or with $(Cut)$) or $\Psi_1$ is not active on the left-hand side of $\Gamma_2, x^1 : \Psi_1 \vdash u^1 : \Psi_2$. What happens here is that we can always permute $(Cut)$ with the rule above it where $\Psi_1$ is not active at least when this rule is not $(\Box R)$ with $\Phi_1$ on the left-hand side of the sequent. (The only nasty case which we shall examine in detail shortly.)

For example in the case of $(\Rightarrow L)$ on the left we have $\Gamma_1 \vdash \Gamma, y^1 : \Phi_{1} \Rightarrow \Phi_{2} \Gamma_{1} \vdash u^1 = u[y/x] \Gamma_{1} \Gamma_{1} = \Phi_{3} \Gamma_{2}$ and we transform:

$$\begin{array}{ll}
\Gamma \vdash v^1 : \Phi & \Gamma_2 \vdash u^1 : \Phi_1 \\
\vdots & \vdots \\
\Gamma, x^1 : \Phi_2 \vdash u^1 : \Phi_3 & \Gamma \vdash v^1 : \Phi_1 \\
\hline
\Gamma, y^1 : \Phi_{1} \Rightarrow \Phi_{2} \vdash u^1 = u[y/x] \Gamma_{1} \Gamma_{1} = \Phi_{3} \Gamma_{2} \vdash u^1 : \Psi_2 \\
\hline
\end{array}$$

Into:

$$\begin{array}{ll}
\Gamma, x^1 : \Phi_2 \vdash u^1 : \Phi_3 & \Gamma_2 \vdash u^1 : \Phi_2 \\
\vdots & \vdots \\
\Gamma \vdash v^1 : \Phi & \Gamma_2 \vdash u^1 : \Phi_1 \\
\hline
\Gamma, x^1 : \Phi_2 \vdash u^1 = u[y/x] \Gamma_{1} \Gamma_{1} = \Phi_{3} \Gamma_{2} \vdash u^1 : \Psi_2 \\
\hline
\end{array}$$

Computationally speaking this rewrites $u^2[u/y/x][v/x]_1$ into $u^2[u/x][y/x]_1$ which is not only valid but in fact vacuously so. Indeed these terms are already equal by the variable naming convention. In fact none
of these transformations do anything on the term level. (They would if we had chosen a calculus of explicit substitutions [ACCL99].)

The only problem occurs when the last rule of (π₂) is (□R)—in particular Γ₁Ψ₁ is inactive on the left-hand side of Γ₂, x₁ : Ψ₁ ⊢ u₂ : Ψ₂. The cut then looks like:

\[
\begin{array}{c}
(\pi_2) \\
(\pi_1) \\
\vdots \\
\Gamma ⊢ u : \Phi \\
\Gamma_1 ⊢ u_1 : \Psi_1 \quad (\Box R) \quad \Gamma, \Gamma', x : \Psi_1 ⊢ u^* : \Box \Phi \\
\hline
\end{array}
\]

\[(\text{Cut})\]  
\[\Gamma_1, \Gamma, \Gamma' ⊢ u \left[ u_1/x \right] : \Box \Phi\]

where Γ is boxed Γ and (unless Γ₁ is boxed) we cannot transform this into the following:

\[
\begin{array}{c}
(\pi_1) \\
\vdots \\
\Gamma_1 ⊢ u_1 : \Psi_1 \quad \Gamma, x : \Psi_1 ⊢ u : \Phi \\
\hline
\end{array}
\]

\[(\text{Cut})\]  
\[\Gamma, \Gamma_1, \Gamma' ⊢ u \left[ u_1/x \right] : \Phi, \Box \Phi\]

which is indeed not a proof at all if Γ₁ is not boxed.

So we have to find another way and examine the last rule we used in (π₁). There are several cases and subcases:

- Either Ψ₁ is inactive on the right of the rule and we just bubble the (Cut) up inside (π₁). This has no effect on the resulting construction: for example if the last rule of (π₁) is (⇒ L)Γ then u₁ has the form v₁[yv₂/z] so the construction before the cut is pushed upwards is u'[v₁[yv₂/z]/x]Γ and the construction afterwards is u'[v₁/x][yv₂/z]Γ which is the same. Because of the variable naming convention. Note that if the last rule of (π₁) is another (Cut)Γ pushing the lower cut upwards does not change the construction either.

- Or Ψ₁ is active on the right of the rule and we again have two cases according to whether Ψ₁ is a boxed formula or not:
  - Case (a) if Ψ₁ is not a boxed formula then x cannot occur free in u since the assumption x : Ψ₁ cannot be used to type u in (π₁). We can therefore transform the proof into:
    \[
    \begin{array}{c}
    (\pi_2') \\
    \vdots \\
    \Gamma ⊢ u : \Phi \\
    (\Box R) \quad \Gamma_1, \Gamma, \Gamma' ⊢ u^* : \Box \Phi \\
    \hline
    \end{array}
    \]

    This suggests the identity u'[u₁/x] ≡ u' when x is not free in u. That is if x is not free in u then it should not be free in u'.

- if Ψ₁ is a boxed formula □Φ₁Γ we can assume that the end sequent of (π₁) contains an assumption of the form x : □Φ₁ on its left-hand side and we can therefore bubble up the (Cut) above the (□R) rule on the right. Notice that the only ways that □Φ₁ can be active on the right-hand side of the end sequent of (π₁) are that the last rule of (π₁) be either (Ax) or (□R). This yields two sub-cases:

  Case (b) The last rule of (π₁) is (Ax)Γ and the proof looks like:

\[
\begin{array}{c}
(\pi_2') \\
\vdots \\
\Gamma, x : □Φ₁ ⊢ u : \Phi \\
(\Box R) \quad \Gamma, \Gamma', x : □Φ₁ ⊢ u^* : \Box \Phi \\
\hline
\end{array}
\]

\[(\text{Ax}) \quad \Gamma_1, y : □Φ₁ ⊢ y : □Φ₁ \quad (\Box R) \quad \Gamma, \Gamma', x : □Φ₁ ⊢ u^* : \Box \Phi \]

\[(\text{Cut}) \quad \Gamma_1, \Gamma, \Gamma', y : □Φ₁ ⊢ u^* \left[ y/x \right] : \Box \Phi \]
\[
\begin{align*}
(Ax) & \quad \Gamma, x : \Phi \vdash x : \Phi \\
(\Rightarrow I) & \quad \Gamma, x : \Phi_1 \vdash u : \Phi_2 \quad \vdash E \quad \Gamma \vdash u : \Phi_1 \Rightarrow \Phi_2 \\
(\Rightarrow E) & \quad \Gamma, \lambda x : \Phi_1 \vdash u : \Phi_2 \\
(\Box I) & \quad \Gamma \vdash u : \Phi \\
(\Box E) & \quad \Gamma, \Gamma' \vdash u' : \Box \Phi \\
& \quad \Gamma \text{ boxed} \\
\end{align*}
\]

Figure 2: Natural Deduction for S4

and we transform it into:

\[
\begin{align*}
(\pi'_2)[y/x] \\
\vdots \\
(\Box R) & \quad \Gamma, y : \Box \Phi_1 \vdash y[x/y] : \Phi \\
\end{align*}
\]

This suggests that the free variables of \(u^i\) should in fact be the same as those of \(u\).

**Case (c)** The last rule of \((\pi_1)\) is \((\Box R)\)\(\Gamma\) and the proof looks like:

\[
\begin{align*}
(\pi'_1) & \quad (\pi'_2) \\
\vdots & \quad \vdots \\
(\Box R) & \quad \Gamma' \vdash v_1 : \Phi_1 \\
(\Box R) & \quad \Gamma, x : \Box \Phi_1 \vdash u : \Phi \\
(Cut) & \quad \Gamma, \Gamma' \vdash u'[v_1/x] : \Box \Phi
\end{align*}
\]

where \(\Gamma'_1\) is a boxed context included in \(\Gamma_1\) and \(u_1 = v_1^i\). We can transform this by bubbling \((Cut)\) on the right-hand sequent yielding:

\[
\begin{align*}
(\pi'_1) & \quad (\pi'_2) \\
\vdots & \quad \vdots \\
(\Box R) & \quad \Gamma' \vdash v_1 : \Phi_1 \\
(\Box R) & \quad \Gamma, x : \Box \Phi_1 \vdash u : \Phi \\
(Cut) & \quad \Gamma, \Gamma' \vdash u[v_1/x] : \Phi \\
(\Box R) & \quad \Gamma, \Gamma' \vdash (u[v_1/x]) : \Box \Phi
\end{align*}
\]

Should we choose \(u^i = u\) as a definition? This would be automatically satisfied.

In short \(\Gamma\) cut-elimination works (although we still have to prove termination of the process) but the \((\Box R)\) rule blocks some cuts going up — at least temporarily.

### 4.2 A First Try

Let’s develop the first naïve way of eliminating cuts \(\Gamma\) and ignore the proviso that some cuts cannot be pushed up \((\Box R)\) right away:

**Theorem 4.1** Let the \(\lambda\mu\)-calculus be the \(\lambda\)-calculus with operations \(*, \cdot, \) with the following reduction rules:
• \((\lambda x \cdot u)v \rightarrow u[v/x]\)
• \(s(u) \rightarrow u\)

Consider the typing rules of Figure 2. Then:

• The reduction rules have the Church-Rosser property.
• All typed terms are strongly normalizing.
• Typable normal forms are constructions for cut-free \(LS_4\) proofs.
• Type-checking and type inferencing is decidable in polynomial time, and every typable term has a most general type.
• Subject reduction fails, i.e. there is a typable term \(u\) such that \(u \rightarrow v\), but \(v\) is not typable.

**Proof:** See Appendix B for the proofs of all but the last claim.

We show that subject reduction fails. Consider \(\lambda x \cdot \lambda y \cdot (\lambda z \cdot z)(xy)\) where \(z : \Box x : B \Rightarrow \Box y : B\).
Its most general type is \((B \Rightarrow \Box A) \Rightarrow B \Rightarrow \Box \Box A\). But it reduces to \(\lambda x \cdot \lambda y \cdot (xy)I\) which is not typable under the assumptions \(x : B \Rightarrow \Box A\) and \(y : B\). In fact, this term \(I\) which is the normal form is not typable at all since to be typable \(x\) and \(y\) have to be of a boxed type (because otherwise they won’t be visible from inside the \(\Box\)). But then \(xy\) cannot be typed.

The termination proof extends to any type system for the pure \(\lambda\)-calculus such that typable terms are strongly normalizing. This includes Girard’s System F [Gir71,FGLT89] [Krivine and Leivant’s AF2 [Kri92] T] or Huet and Coquand’s calculus of constructions [CH87]. On the other hand, the decidability of type-checking is the same as the decidability of the type-checking problem for the pure \(\lambda\)-calculus fragment of the typed language; this is because the typing theory for the \(\lambda++\)-calculus is a conservative extension of that for the pure \(\lambda\)-calculus.

Notice that the typing rules of Figure 2 are nothing but a reformulation of \(LS_4\) in natural deduction style. The introduction rules \((\Rightarrow I)\) and \((\Box I)\) are directly the right rules \((\Rightarrow R)\) and \((\Box R)\) respectively. The elimination rules \((\Rightarrow E)\) and \((\Box E)\) are obtained from the respective left rules \((\Rightarrow L)\) and \((\Box L)\) by means of \((\exists u)\). Hence any construction in \(LS_4\) is a typable \(\lambda++\)-term.

Because of the failure of subject reduction, \(I\) cannot be concluded directly that the cut-elimination process terminates; indeed, reduction may terminate on an unjustifiable term. This is paradoxical because not only does cut-elimination terminate but in fact the \(\lambda++\)-reduction rules go too far in that they end up producing terms that are the constructions of no \(LS_4\) proof at all! This is why we shall now investigate more complicated but more meaningful ways of defining \(u^*\) for any given \(u\).

### 4.3 A Solution Based on Bierman and de Paiva’s Proposal

This morphology of Section 4.1 is the following: \(u^*\) should behave as \(u\Gamma\) in as much as \(u^*\) must rewrite to \(u\Gamma\) but it cannot be built from \(u\) by just applying some operators to \(u\). In fact, it appears that \(u^*\) should operate much like a barrier preventing all substitutions that we want to apply to \(u^*\) to be propagated to \(u\) itself. Parts of these substitutions may be kosher though — they might introduce variables that are of boxed types only — \(\Gamma\) and we can allow these substitutions but not the others. To continue their route down the term.

We can now understand \(u^*\) as being not \(u\Gamma\) but something like \(v_{p_1 ::= x_1, \ldots, p_n ::= x_n}^*\) \(\Gamma\) where \(x_1, \ldots, x_n\) are the free variables of \(u\). The explicit substitution \([x_1 ::= x_1, \ldots, x_n ::= x_n]\) first binds all free variables of \(u\) (as left-hand sides of \(:=\) ) and then binds them to terms that happen to be \(x_1, \ldots, x_n\). This way, the only place where we can substitute terms for free variables is in the explicit substitution part of \(v_{p_1 ::= x_1, \ldots, p_n ::= x_n}^*\) \(\Gamma\). Therefore blocking substitutions from affecting \(u\) itself. Indeed, \(u^*[u_1/x_i]\) now means 
\[v_{p_1 = x_1, \ldots, p_n = x_n}^*\] which is different from \(u[u_1/x_i]\) and uniqueness.

This is the solution of Bierman and de Paiva [BdP95] (except that they also consider sum types i.e. disjunctions). Their notation for \(v_{p_1 = x_1, \ldots, p_n = x_n}^*\) is \(\text{box } u\) with \(x_1, \ldots, x_n\) for \(x_1, \ldots, x_n\) \(\Gamma\) where the variables after
the for keyword are considered bound \( \Gamma \) and \( u \) is written unbox \( u \Gamma \) with the following sole rewrite rule on \( \text{box/unbox} \):

\[
\text{unbox (box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n \Rightarrow u[v_1/x_1, \ldots, v_n/x_n]
\]

To respect Barendregt’s naming convention \( u' \) is in fact encoded as follows (we make this a definition \( \Gamma \) since we shall use it in the rest of this section):

**Definition 4.1** For any term \( u \), with free variables \( x_1, \ldots, x_n \), we define \( u' \) as:

\[
\text{box } u[y_1/x_1, \ldots, y_n/x_n] \text{ with } x_1, \ldots, x_n \text{ for } y_1, \ldots, y_n
\]

where \( y_1, \ldots, y_n \) are \( n \) pairwise distinct fresh variables.

The variables \( y_{i\Gamma} \leq i \leq n_{\Gamma} \) are the bound variables of the construct \( \Gamma \) and are subject to \( \alpha \)-renaming (outside \( \text{fv}(u) \)). Notice that the general form:

\[
\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n
\]

then encodes \( u'[v_1/x_1, \ldots, v_n/x_n] \).

However this calculus is not satisfactory. Indeed the latter rule deals with the case of a cut-elimination in the \((\square R)/(\square L)\) case with the cut formula active on both sides (see Section 4.1.1). But other cut-eliminations are not handled. Then we have two solutions.

Either we trade the \((\square R)\) rule as Bierman and de Paiva do for the following compound rule \((\square I)\):

\[
\Gamma \vdash v_1 : \square \Phi_1 \ldots \Gamma \vdash v_n : \square \Phi_n \quad x_1 : \square \Phi_1, \ldots, x_n : \square \Phi_n \vdash u : \Phi
\]

but this rule does not respect the subformula property. Therefore it is not acceptable as a non-cut rule in a sequent calculus. (It is fine for natural deduction though.) To put it another way, normal proofs in their system are only normal in a weak sense and may still contain hidden cuts; this makes the system unsuitable for \( \text{Isay}\) automated deduction where the subformula property is crucial.

We are forced to admit that the previous rule is just the following special combination of \((\square R)\) and \((Cuf)\):

\[
\begin{align*}
\Gamma \vdash v_n : \square \Phi_n & \quad (\square R) \\
\Gamma \vdash v_1 : \square \Phi_1, \ldots, x_n : \square \Phi_n \vdash u : \Phi & \quad (Cuf) \\
\end{align*}
\]

and then Bierman and de Paiva’s rule is not enough to eliminate all cuts since for example \((\square R)/(\square R)\) cuts (case (c) in Section 4.1.2) are not eliminated: the prototypical example is \( \text{box } x \) with \( \{ \text{box } u \text{ with } \epsilon \text{ for } \epsilon \} \) for \( x \) (where we write \( \epsilon \) for the empty list of terms or of variables) which is normal in their calculus but can only be the construction for a proof containing a \((\square R)/(\square R)\) cut.

To repair this we are led to add the following rewrite rule:

\[
\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n \\
\Rightarrow \text{box } u[v'_1/x_1] \text{ with } v_1, \ldots, v_{n-1}, w_1, \ldots, w_m, v_{n+1}, \ldots, v_n \text{ for } x_1, \ldots, x_{n-1}, y_1, \ldots, y_m, x_{n+1}, \ldots, x_n
\]

when \( v_1 \) has the form \( \text{box } v'_1 \) with \( w_1, \ldots, w_n \) for \( y_1, \ldots, y_m \) (we say that \( v_1 \) is a box-term).

But cases (a) and (b) also pose problems of their own. Case (a) demands that all the free variables of \( u' \) are free in \( u \) for at least that \( u' \) rewrites to some term having no more free variables than \( u \). This suggests adding a garbage collection rule:

\[
\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n \\
\Rightarrow \text{box } u \text{ with } v_1, \ldots, v_{n-1}, v_{n+1}, \ldots, v_n \text{ for } x_1, \ldots, x_{n-1}, x_{n+1}, \ldots, x_n
\]
whenever the proof. That is, for any permutation \( \pi \) of \( \{1, \ldots, n\} \) the substitution \([v_1/x_1, \ldots, v_n/x_n]\) is identical to \([v_{\pi(1)}/x_{\pi(1)}, \ldots, v_{\pi(n)}/x_{\pi(n)}]\). This transposes to considering the terms:

\[
\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n
\]

and

\[
\text{box } u \text{ with } v_{\pi(1)}, \ldots, v_{\pi(n)} \text{ for } x_{\pi(1)}, \ldots, x_{\pi(n)}
\]
as being equivalent. This is consistent with the previous rewrite rules.

To recap, we define:

**Definition 4.2 (\( \lambda S_4 \))** The \( \lambda S_4 \)-calculus is defined by the following syntax, where \( x, y, z, \ldots \) denote variables taken from an infinite set \( \mathcal{V} \). Terms \( s, t, u, v, w, \ldots \) are elements of the language \( T \) defined by:

\[
T ::= \mathcal{V} \mid \lambda v. T \mid TT \mid \text{unbox } T \mid \text{box } T \text{ with } T, \ldots, T \text{ for } \mathcal{V}, \ldots, \mathcal{V}
\]

where \( x_i \) is not free in \( u \). (This is the same rule as the above except the side-condition is on \( x_i \) and \( u \) instead of on \( v_i \).)

Case (b) demands that all the free variables of \( u \) are also free in \( u' \). This places the restriction on terms of the form box \( u \) with \( v_1, \ldots, v_n \) for \( x_1, \ldots, x_n \) that any free variable of \( v \) is some \( x_i \Gamma \leq i \leq n \). This restriction is then preserved by the rewrite rules.

Because box encodes a substitution inside the language itself as an association between the terms after with and the variables after for we also expect a few other properties of substitutions to hold of box. The first one which we need for proving that every normal form of the calculus represents some cut-free LS4 proof is related to the contraction rule in sequent calculus:

\[
\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n \sim \text{box } u \text{ with } v_{\pi(1)}, \ldots, v_{\pi(n)} \text{ for } x_{\pi(1)}, \ldots, x_{\pi(n)}
\]

for any permutation \( \pi \) of \( \{1, \ldots, n\} \)

Figure 3: Reduction in the \( \lambda S_4 \)-Calculus
Figure 4: Typing Rules for the $\lambda_{S4}$-Calculus

where the \textit{with} part of a box term must have exactly as many terms as the \textit{for} part has variables, and all variables in the \textit{for} part are pairwise distinct. Moreover, we require that any term

$$\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n$$

is such that $\text{fv}(u) \subseteq \{x_1, \ldots, x_n\}$, and $\text{fv}(v_i) \cap \{x_1, \ldots, x_n\} = \emptyset$ for every $i$, $1 \leq i \leq n$.

The rewrite rules are given in Figure 3. The relation $\sim$ is defined as the smallest congruence relation verifying the commuting conversion equation in Figure 3 and such that if $u$ and $v$ are $\alpha$-convertible, then $u \sim v$. Unless stated otherwise, we always consider $\lambda_{S4}$-terms modulo $\sim$.

The typing rules are given in Figure 4.

We shall sometimes write $\text{box } u$ with $\sigma$ instead of the more cumbersome $\text{box } u$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$ where $\sigma = \{v_1/x_1, \ldots, v_n/x_n\}$.

The rewrite rules are precisely the above namely those of Bierman and de Paiva plus the ones necessary to model cut-elimination fully. The typing rules which can also be read as a natural deduction system for $S4$ are exactly those given in [BdP95]; we have included them here for completeness.

Notice that the rewrite rules preserve the well-formedness constraints on terms and are consistent with the equivalence $\sim$ on terms ($\alpha$-conversion and commuting conversion) so the calculus is well-defined.

5 Properties of the $\lambda_{S4}$-Calculus

The $\lambda_{S4}$-calculus is then (almost) perfect. We prove the following in Section 5.1:

\textbf{Theorem 5.1} The properties of the $\lambda_{S4}$-calculus are:

- The rewrite rules have the Church-Rosser property.
- Subject reduction holds.
- All typed terms are strongly normalizing.
- Typable normal forms are constructions for cut-free $\text{LS4}$ proofs.
- Type-checking and type inferencing are decidable in polynomial time, and every typable term has a most general type.

In Section 5.2 we shall discuss why the $\lambda_{S4}$-calculus is in fact not quite satisfactory.
5.1 Proofs of the Properties

The most natural way to prove that the $\lambda_{54}$-calculus is confluent is to first prove the finiteness of developments $\Gamma$ as in [Bar84]11.2 for the case of the pure $\lambda$-calculus.

**Definition 5.1 ($\lambda_{54}$)** We define the $\lambda_{54}$-calculus as follows.

The terms $u, v, w, \ldots$ are variables $x, y, z, \ldots$, applications $uv$, abstractions $\lambda x \cdot u$ and primed redexes $\lambda^i x \cdot u \cdot v$ (we then consider all free occurrences of $x$ in $u$ as bound), evaluations $\text{unbox } u$ and quotations $\text{box } u$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$ (then, all free occurrences of $x_i$, $1 \leq i \leq n$, are considered bound in the quotation). These terms are subject to the same provisos (see Definition 4.2) and variable naming convention than $\lambda_{54}$-terms.

Reduction in the $\lambda_{54}$-calculus is defined as in $\lambda_{54}$, except that $(\beta)$ is replaced by:

$$(\beta') (\lambda^i x \cdot u \cdot v) \rightarrow u[v/x]$$

The erasing transformation $E$ from $\lambda_{54}^i$ to $\lambda_{54}$ consists in erasing all prime symbols from $\lambda^i$-redexes (i.e., $E((\lambda^i x \cdot u) \cdot v) = (\lambda x \cdot E(u)) \cdot E(v)$, in particular.)

Notice that primed redexes are not considered as applications but rather as entirely new expressions; think of $(\lambda^i x \cdot u) \cdot v$ as another notation for $\text{let } x = v \text{ in } u$.

Our aim is to show that the $\lambda_{54}$-calculus terminates. To do so, we define an appropriate notion of weighting of variables and terms:

**Definition 5.2** We define the $\lambda_{54}^{*}$-calculus as the following extension of the $\lambda_{54}$-calculus.

The terms are as in the $\lambda_{54}$-calculus, except that each variable can be either a weighted variable $x^k$, where $k$ is a positive integer, or a proxy variable $x, y, \ldots$

More precisely, the $\lambda_{54}^{*}$-terms $u, v, w, \ldots$ are proxy variables $x, y, \ldots$, weighted variables $x^k, y^k, z^k, \ldots$ ($k > 0$), applications $uv$, abstractions $\lambda x \cdot u$ and primed redexes $\lambda^i x \cdot u \cdot v$ (we then consider all free occurrences of $x^k$, for any $k > 0$, in $u$ as bound), evaluations $\text{unbox } u$ and quotations $\text{box } u$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$ (then, for each $i$, all free occurrences of $x_i^k$, for any $k > 0$, or of $x$ in $u$ are considered bound in the $\text{box-term}$.)

These terms are subject to the same provisos and variable naming convention than $\lambda_{54}$-terms. Moreover, we impose that no variable $x$ occurs both as a proxy and as a weighted variable $x^k$.

Well-formed $\lambda_{54}$-terms are those $\lambda_{54}^{*}$-terms such that every occurrence of a proxy variable is bound by some $\text{box-expression}$.

Substitution in the $\lambda_{54}^{*}$-calculus is defined so that weights and proxy status are ignored (i.e., in particular $x^k[u/x] = x^k[u/x] = u, x^k[u/y] = x^k$ and $x[y/x] = x$ if $x \neq y$).

We also define a related notion of weight-preserving renaming $u\{x/y\}$, so that $y^k\{x/y\} = x^k, y\{x/y\} = x$, and $z^k\{x/y\} = z^k, z\{x/y\} = z$ if $z \neq y$.

Reduction in the $\lambda_{54}^{*}$-calculus is then defined as in $\lambda_{54}^{*}$, except for the change of meaning of substitution, and for the following reformulation of the rules ($\text{box}$) and ($\text{extract}$):

$$\begin{align*}
\text{(box')} & \quad \text{box } u \text{ with } v_1, \ldots, v_i, \ldots, v_n \text{ for } x_1, \ldots, x_i, \ldots, x_n \\
& \quad \rightarrow \text{box } u[\text{box } v \text{ with } z_1, \ldots, z_m \text{ for } y_1, \ldots, y_m/x_i] \\
& \quad \quad \text{with } v_1, \ldots, v_i-1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \\
& \quad \quad \text{for } x_1, \ldots, x_i-1, z_1, \ldots, z_m, x_{i+1}, \ldots, x_n
\end{align*}$$

if $v_i = \text{box } v$ with $w_1, \ldots, w_m$ for $y_1, \ldots, y_m$, and where $z_1, \ldots, z_m$ are in fresh pairwise distinct proxy variables,

$$\begin{align*}
\text{(extract')} & \quad \text{box } u \text{ with } \ldots, v_i, \ldots, v_j, \ldots \text{ for } \ldots, x_i, \ldots, x_j, \ldots \\
& \quad \rightarrow \text{box } u[x_i/x_j] \text{ with } \ldots, v_i, \ldots, v_j-1, v_{j+1}, \ldots \text{ for } \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots
\end{align*}$$

if $v_i = v_j, i \neq j$.

The erasing transformation $E_0$ from $\lambda_{54}^{*}$ to $\lambda_{54}$ consists in erasing all weights and proxy statuses (i.e., in particular $E_0(x^k) = E_0(x) = x$).
Given a term \( u \Gamma \Gamma \) if \( x \) only occurs as a proxy \( x \) in \( u \) then we shall say that \( x \) itself is a proxy variable; otherwise \( x \) by our conventions \( x \) can only occur weighted in \( u \Gamma \) and we shall say that \( x \) itself is a weighted variable.

Notice that the formulation of rule (box) above is equivalent to that in Figure 3 modulo \( a \)-renaming (and \( E_{b} \)-erasing). The idea in using proxy variables is that in the formulation for rule (box) above the proxies \( z_{j} \Gamma \) \( 1 \leq j \leq m \) are not meant to have weights of their own but rather to represent \( u_{j} \) to which they are bound by the outer box term.

We define the \textit{weight} of a well-formed \( \lambda_{54}^{*} \)-term as the sum of the weights of variables occurring (free or bound) in \( \Gamma \) where the weight of \( x^{i} \) is \( k \Gamma \) and the weight of \( x \) is recursively defined as the weight of the term for which it is a proxy. Formally:

**Definition 5.3** For every \( \lambda_{54}^{*} \)-term \( u \), for every function \( \omega \) mapping proxy variables to positive integers, define the weight \( W(u, \omega) \) by structural induction as follows:

- \( W(x^{k}, \omega) = k \);
- \( W(\langle x \cdot u \rangle, \omega) = W(u, \omega) \);
- \( W(\lambda x \cdot u, \omega) = W(\text{unbox } u, \omega) = W(u, \omega) \);
- \( W(\text{box } u \text{ with } v_{1}, \ldots, v_{n} \text{ for } x_{1}, \ldots, x_{n}) = W(u, \omega') + \sum_{i=1}^{n} W(v_{i}, \omega) \), where \( \omega' \) maps every proxy variable \( x \) to \( \omega(x) \), except if \( x = x_{i} \) for some \( i \), \( 1 \leq i \leq n \), in which case \( \omega'(x) = W(v_{i}, \omega) \).

It is easy to see that the weight of a well-formed \( \lambda_{54}^{*} \)-term \( u \) is independent of the particular function \( \omega \) that we choose; we shall write it \( W(u) \). Definition 5.3 then also defines the weight of every sub-term \( v \) of a given well-formed \( \lambda_{54}^{*} \)-term \( u \) relatively to \( \omega \) which we shall write \( W_{\omega}(v) \).

Notice also that because addition is associative and commutative \( \lambda_{54}^{*} \)-Definition 5.3 makes sense modulo the commutative conversion \( \sim \).

As in [Bar84] we can see \( \lambda_{54}^{*} \)-terms as \( \lambda_{54}^{*} \)-terms plus an additional mapping from occurrences of variables to either a positive integer \( k \) (to represent \( x^{i} \)) or to a special proxy token (to represent \( x \)). Such a mapping is called a \textit{weighting}; terms that have the same \( E_{b} \)-erasures only differ by their weightings.

**Definition 5.4** We say that the \textit{weighting of a well-formed \( \lambda_{54}^{*} \)-term} \( t \) is decreasing if and only if:

- for every subterm of \( t \) of the form \( \langle x \cdot u \rangle \), for every free occurrence \( x^{k} \) of \( x \) in \( u \), \( k > W_{\omega}(v) \);
- and for every subterm of \( t \) of the form:

\[
\text{unbox } u \text{ with } v_{1}, \ldots, v_{n} \text{ for } x_{1}, \ldots, x_{n}
\]

for every free occurrence \( x^{i} \) of the non-proxy variable \( x_{i} \) in \( u \), where \( 1 \leq i \leq n \), \( k > W_{\omega}(v_{i}) \).

**Lemma 5.2** For every \( \lambda_{54}^{*} \)-term \( u_{0} \), \( u_{0} \) admits a decreasing weighting; that is, there is a well-formed \( \lambda_{54}^{*} \)-term \( u \) such that \( E_{b}(u) = u_{0} \), and such that the weighting of \( u \) is decreasing.

**Proof:** Number the occurrences of variables in \( u_{0} \) from right to left giving the \( n \)th occurrence the weight \( 2^{n} \) so that for every occurrence \( x^{k} \) of a variable \( x \) \( k \) is greater than the sum of all weights of occurrences of variables occurring to its right. As \( u \) has no proxy variables the weight \( W_{\omega}(v) \) of any subterm \( v \) of \( u \) is just the sum of the weights \( k \) of variable occurrences \( x^{k} \) in \( v \Gamma \) and the result is clear. (This is as in [Bar84].)

**Lemma 5.3** Let \( u \) and \( v \) be two \( \lambda_{54}^{*} \)-terms, such that \( u \) is well-formed, the weighting of \( u \) is decreasing and \( v \) reduces to \( v \) in one step. Then:

(i) \( v \) is well-formed.

(ii) \( W(u) \geq W(v) \), and \( W(u) > W(v) \) unless the contracted redex is of the form

\[
\text{unbox } (\text{box } s \text{ with } t_{1}, \ldots, t_{n} \text{ for } z_{1}, \ldots, z_{n})
\]

where every \( z_{i} \), \( 1 \leq i \leq n \), is a proxy variable.
(iii) The weighting of \( v \) is decreasing.

**Proof:** Let \( \Delta_1 \) be the contracted redex in \( u \).

(i) is clear; observe that all the fresh proxy variables \( z_1 \Gamma \ldots \Gamma z_m \) in rule \((\text{box}^i)\) are indeed bound by the outer box-term.

(ii) If \( \Delta_1 \) is a \((\beta')\) redex \((\lambda' x \cdot s)\Gamma \) then zero one or more occurrences \( x^k \) of one variable \( x \) in \( s \) are replaced by a term \( t \) of strictly smaller weight. (Because \( u \) is well-formed \( \lambda' \)-bound variable \( x \) is not a proxy.) Even when \( x \) has no occurrence in \( s \) the weight still decreases strictly because the vanishing term \( t \) has strictly positive weight.

If \( \Delta_1 \) is an \((\text{unbox})\) redex:

\[
\text{unbox} \ (\text{box} \ s \ with \ t_1, \ldots, t_n \ for \ x_1, \ldots, x_n)
\]

then zero one or more occurrences of zero one or more variables \( x_1 \Gamma \ldots \Gamma x_n \) are replaced by terms \( v_1 \Gamma \ldots \Gamma v_n \) of no greater weights. So the weight is non-increasing in this case. Moreover if some \( x_i \Gamma \ 1 \leq i \leq n \) is a non-proxy variable then by the same argument as for \((\beta')\)-redexes the weight decreases strictly.

If \( \Delta_1 \) is a \((\text{box}^i)\) redex:

\[
\text{box} \ s \ with \ t_1, \ldots, t_i, \ldots, t_n \ for \ x_1, \ldots, x_i, \ldots, x_n
\]

with \( t_i = \text{box} t \ with \ w_1, \ldots, w_m \ for \ y_1, \ldots, y_m \) then we have two cases.

- Either \( x_i \) is a weighted variable \( \Gamma \) and:
  
  \[
  s[\text{box} \ t \ with \ z_1, \ldots, z_m \ for \ y_1, \ldots, y_m]/x_i
  \]

  whose weight in \( u \) (or \( v \)) is exactly the same as that of \( t_i \Gamma \) i.e. is strictly less than the weight of any replaced occurrence of \( x_i \). So the weight of

\[
W(u) > W(v).
\]

- Or \( x_i \) is a proxy \( \Gamma \) and on the one hand the weight of \( s[\text{box} \ t \ with \ z_1, \ldots, z_m \ for \ y_1, \ldots, y_m]/x_i \) in \( v \) is exactly the same as that of \( s \) in \( u \); and on the other hand \( \Gamma t_i \) is replaced by \( w_1 \Gamma \ldots \Gamma w_m \) in the with part of \( \Delta_1 \) so that again the weight decreases strictly.

If \( \Delta_1 \) is a \((\text{gc})\) redex \( \Gamma \) then the weight decreases strictly because one term of positive weight vanishes.

If \( \Delta_1 \) is a \((\text{contract}^i)\) redex \( \text{box} \ u \ with \ v_1, \ldots, v_i, \ldots, v_n \ for \ x_1, \ldots, x_i, \ldots, x_n \) \( \Gamma \) with \( v_i = v_j \Gamma \) then contracting it means replacing \( u \) by \( u\{x_i/x_j\} \Gamma \) whose weight does not change \( \Gamma \) and deleting \( v_j \Gamma \) which makes the overall weight decrease.

(iii) Let \( \Delta_1 \) be a \( \lambda' \)-redex or a box term occurring in \( v \). \( \Delta_0 \) is the residual of some \( \lambda' \)-redex (resp. of some box term) \( \Delta_2 \) in \( u \). We prove the claim by case analysis on the respective positions of \( \Delta_1 \) and \( \Delta_2 \) in \( u \).

The only non-trivial cases are:

- If \( \Delta_2 \) is \((\lambda' x \cdot u')^{v'} \) (resp. \( \text{box} u' \ with \ v_1, \ldots, v_n \ for \ x_1, \ldots, x_n \) \( \Gamma \) with \( x = x_i \) being a non-proxy variable for some arbitrary \( i \ 1 \leq i \leq n \) in which case we let \( v' = v_i \) and \( \Delta_1 \) lies inside \( v' \Gamma \) then by (i) \( \Delta_1 \) is contracted to a term with weight at most that of \( \Delta_2 \); therefore \( v' \) reduces during this contraction to a term with at most the same weight \( \Gamma \) and every occurrence \( x^k \) of \( x \) of \( u' \) remains of greater weight than that of the contracted \( v' \)
If $\Delta_1$ is $(\lambda' x \cdot u')v'$ (resp. $\box x u'$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$) with $x = x_i$ a non-proxy variable for some arbitrary $\iota\Gamma \leq i \leq n\Gamma$ in which case we let $v' = v_i$) and $\Delta_2$ lies inside $u'$. Let $\Delta_2$ be $(\lambda' x' \cdot u)v''$ (resp. $\box x u$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$) with $x' = x'_j$ and $v'' = v'_j$ for some arbitrary $\j\Gamma \leq j \leq n$. Then contracting $\Delta_1$ replaces $\Delta_2$ by $(\lambda' x' \cdot u''[v'/x]) (v''[v'/x])$ (resp. $\box x u''$ with $v'_1[v'/x], \ldots, v'_n[v'/x]$ for $x_1, \ldots, x'_n$).

If $x$ is a proxy (meaning that $\Delta_1$ must be a box-term) then $W_\Gamma (v''[v'/x]) = W_\Gamma (u')$ by definition of the weight (the weight of the proxy is the weight of the term which is substituted for it). If $x$ is a weighted variable then because the weighting of $u$ is decreasing, all the occurrences of $x$ have lesser weight than $v$ so $W_\Gamma (v''[v'/x]) \leq W_\Gamma (u')$. In any case $W_\Gamma (v''[v'/x]) \leq W_\Gamma (u') \Gamma$ and because the weighting of $u$ is decreasing for every occurrence $x^k$ of $x'$ in $u''[v'/x]$ (equivalently $W_\Gamma (u') \Gamma$ $k > W (u') \geq W (v''[v'/x])$ so the weighting of $v$ remains decreasing.

$\square$

**Lemma 5.4** In $\lambda_5'$, every sequence of (unbox) steps is finite.

**Proof:** Define the following translation from $\lambda_5'$ (without the commutative conversion rule) to the $\lambda'$-calculus: if $\mathrm{The}$ language built on variables, applications, $\lambda$-abstractions and $\lambda'$-reduces $\Gamma$ with $\beta$ as sole reduction rule:

- $I(x) = x$;
- $I(uv) = I(u) I(v) I(\lambda x \cdot u) = \lambda x \cdot I(u) I(I(\lambda' x \cdot u)v) = (\lambda' x \cdot I(u)) I(v)$;
- $I(\unbox xu) = (\lambda' x \cdot x) I(u)$;
- $I(\box xu$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n) = (\lambda' x_1 \cdot \ldots \cdot \lambda' x_n \cdot I(u)) I(v_1) \ldots I(v_n)$.

If $u \rightarrow v$ by application of the (unbox) rule $\Gamma$ then $I(u) \rightarrow^+ I(v)$ in $\lambda'$ (by $n + 1$ applications of rule $\beta$). As all developments in the $\lambda$-calculus are finite, all reductions in $\lambda'$ are terminating. So every sequence of (unbox) steps in $\lambda_5'$ (with the commutative conversion rule) must be finite.

If $u_1 \sim u'_1 \rightarrow u_2 \sim u'_2 \rightarrow \ldots \rightarrow u_k$ is any sequence of (unbox) steps in $\lambda_5'$ (with the commutative conversion rule) $\Gamma$ notice that we can always postpone the application of the commutative conversion rule $\Gamma$ so as to obtain a reduction sequence $u''_1 \rightarrow u''_2 \rightarrow \ldots \rightarrow u''_k$ of the same length $\Gamma$ where for each $i\Gamma \leq i \leq k\Gamma$ $u''_i \sim u_i$ (and in fact $u''_1 = u_1$). The latter is a sequence of (unbox) steps in the calculus without the commutative conversion rule $\Gamma$ so its length is bounded by some function $b(u_1)$. This bound is therefore also a bound on the initial sequence from $u_1$ to $u_k$: the claim is proved.

It follows that all developments in $\lambda_5'$ are finite, that is:

**Theorem 5.5** The notion of reduction in $\lambda_5'$ is strongly normalizing.

**Proof:** Let $u_1$ be a $\lambda_5'$-term. By Lemma 5.2 there is a $\lambda_5^{*'}$-term $u$ such that $E(u) = u_1$ and the weighting of $u$ is decreasing. Consider an arbitrary reduction in $\lambda_5'$ starting from $u_1 \Gamma$ and lift it to a corresponding reduction in $\lambda_5^{*'}$ starting from $u$ (replacing rules (box) and (contraction) by (box') and (contraction') respectively). For each term $v$ in the sequence $\Gamma$ let $W_\Gamma (v)$ be the couple $(W_\Gamma (v), \nu (v))$ where $\nu (v)$ is the greatest length of a sequence of (unbox) steps that can be applied from $E_\Gamma (v)$ (which is finite by Lemma 5.4). By Lemma 5.3 $W_\Gamma (v)$ decreases strictly in the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$ which is well-founded; therefore the reduction must be finite.

$\square$

**Lemma 5.6** The $\lambda_5'$-calculus is confluent.

**Proof:** By Theorem 5.5 it is enough to prove that it is locally confluent. The critical pairs are as follows (we won't mention the variable naming convention again):

- If $\Delta_1$ is $(\lambda' x \cdot u')v'$ (resp. $\box x u'$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n\Gamma$ with $x = x_i$ a non-proxy variable for some arbitrary $\iota\Gamma \leq i \leq n\Gamma$ in which case we let $v' = v_i$) and $\Delta_2$ lies inside $u'$. Let $\Delta_2$ be $(\lambda' x' \cdot u)v''$ (resp. $\box x u$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$) with $x' = x'_j$ and $v'' = v'_j$ for some arbitrary $\j\Gamma \leq j \leq n$. Then contracting $\Delta_1$ replaces $\Delta_2$ by $(\lambda' x' \cdot u''[v'/x]) (v''[v'/x])$ (resp. $\box x u''$ with $v'_1[v'/x], \ldots, v'_n[v'/x]$ for $x_1, \ldots, x'_n$).

If $x$ is a proxy (meaning that $\Delta_1$ must be a box-term) then $W_\Gamma (v''[v'/x]) = W_\Gamma (u')$ by definition of the weight (the weight of the proxy is the weight of the term which is substituted for it). If $x$ is a weighted variable then because the weighting of $u$ is decreasing, all the occurrences of $x$ have lesser weight than $v$ so $W_\Gamma (v''[v'/x]) \leq W_\Gamma (u')$. In any case $W_\Gamma (v''[v'/x]) \leq W_\Gamma (u') \Gamma$ and because the weighting of $u$ is decreasing for every occurrence $x^k$ of $x'$ in $u''[v'/x]$ (equivalently $W_\Gamma (u') \Gamma$ $k > W (u') \geq W (v''[v'/x])$ so the weighting of $v$ remains decreasing.

$\square$
• Between (unbox) and (box): assume that \( v_i = \text{box} u \) with \( w_1, \ldots, w_m \) for \( y_1, \ldots, y_m \) then:

\[
\text{unbox} (\text{box} u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n)
\]

reduces in one step to:

\[
u[v_1/x_1, \ldots, v_n/x_n]
\]

by (unbox). But also to:

\[
\text{unbox} (\text{box} u[\text{box} v \text{ with } z_1, \ldots, z_m \text{ for } y_1, \ldots, y_m]/x_i]
\]

with \( v_1, \ldots, v_{i-1}, w_1, \ldots, w_m, v_{i+1}, \ldots, v_n\) for \( x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n \)

by (box). By one application of (unbox) the latter reduces to the former since \( x_i \) is not free in \( u \).

• Between (unbox) and (extract): when \( v_i = v_j \) for \( i \neq j \)

\[
\text{unbox} (\text{box} u \text{ with } \ldots, v_i, \ldots, v_j, \ldots \text{ for } \ldots, x_i, \ldots, x_j, \ldots)
\]

reduces in one step either to:

\[
u[\ldots, v_i/x_i, \ldots, v_j/x_j, \ldots]
\]

or to:

\[
\text{unbox} (\text{box} u[x_i/x_j] \text{ with } \ldots, v_i, \ldots \text{ for } \ldots, x_i, \ldots)
\]

But the latter reduces in one step by (unbox) to \( u[x_i/x_j][\ldots, v_i/x_i, \ldots, v_j/x_j, \ldots] \) which is exactly the former since \( v_i = v_j \).

• Between (box) and itself. If \( i \neq j \) and:

\[
\begin{align*}
v_i &= \text{box} u' \text{ with } w'_1, \ldots, w'_m, \text{ for } x'_1, \ldots, x'_m, \\
v_j &= \text{box} u'' \text{ with } w''_1, \ldots, w''_m, \text{ for } x''_1, \ldots, x''_m
\end{align*}
\]

and

\[
\begin{align*}
\text{box} u \text{ with } v_i, v_j, \ldots \text{ for } x_i, x_j, \ldots
\end{align*}
\]

reduces in one step either to:

\[
\text{box} u[\text{box} v' \text{ with } z'_1, \ldots, z'_m \text{ for } x'_1, \ldots, x'_m]/x_i
\]

with \( w'_1, \ldots, w'_m, \ldots, v_j, \ldots \)

for \( z'_1, \ldots, z'_m, \ldots, x_j, \ldots \)
between by contract site with where renaming. In the last case (k = i) the term above:

\[
\mathbf{box} u with \ldots, v_i, \ldots, v_j, \ldots for \ldots, x_i, \ldots, x_j, \ldots
\]

where:

\[
v_i = v_j = \mathbf{box} v with w_1, \ldots, w_m for x_1, \ldots, x_m
\]

contracts either to:

\[
\mathbf{box} u[\mathbf{box} v with z_1, \ldots, z_m for x_1, \ldots, x_j]/x_i
\]

with \ldots, w_1, \ldots, w_m, \ldots, v_i, \ldots

for \ldots, z_1, \ldots, z_m, \ldots, x_j, \ldots

by (box) or to:

\[
\mathbf{box} u[x_i/x_j] with \ldots, v_i, \ldots for \ldots, x_i, \ldots
\]

by (contrace).

or to:

\[
\mathbf{box} u[(\mathbf{box} v' with z'_1, \ldots, z'_{m'} for x'_1, \ldots, x'_{m'})/x_j]
\]

with \ldots, v_i, \ldots, w'_1, \ldots, w'_{m'}, \ldots

for \ldots, x_i, \ldots, z'_1, \ldots, z'_{m'}, \ldots

Both reduce by one application of box to:

\[
\mathbf{box} u[\mathbf{box} v' with z'_1, \ldots, z'_{m'} for x'_1, \ldots, x'_{m'}/x_i; (\mathbf{box} v'' with z''_1, \ldots, z''_{m''} for x''_1, \ldots, x''_{m''})/x_j]
\]

with \ldots, w'_1, \ldots, w'_{m'}, \ldots, w''_1, \ldots, w''_{m''}, \ldots

for \ldots, z'_1, \ldots, z'_{m'}, \ldots, z''_1, \ldots, z''_{m''}, \ldots

- Between (box) and (gc). If i ≠ j then

\[
v_i = \mathbf{box} v with w_1, \ldots, w_m for x_1, \ldots, x_m
\]

and x_j is not free in u then:

\[
\mathbf{box} u with \ldots, v_i, \ldots, v_j, \ldots for \ldots, x_i, \ldots, x_j, \ldots
\]

reduces in one (box) step either to:

\[
\mathbf{box} u[(\mathbf{box} v with z_1, \ldots, z_m for x_1, \ldots, x_m)/x_i]
\]

with \ldots, w_1, \ldots, w_m, \ldots, v_j, \ldots

for \ldots, z_1, \ldots, z_m, \ldots, x_j, \ldots

or to:

\[
\mathbf{box} u with \ldots, v_i, \ldots, v_j, \ldots for \ldots, x_i, \ldots, x_j, \ldots
\]

by (gc). By applying (gc) to the former and (box) to the latter we get the same term:

\[
\mathbf{box} u[(\mathbf{box} v with z_1, \ldots, z_m for x_1, \ldots, x_m)/x_i]
\]

with \ldots, w_1, \ldots, w_m, \ldots, v_j, \ldots

for \ldots, z_1, \ldots, z_m, \ldots, x_j, \ldots

- Between (box) and (ctrace). Consider a term:

\[
\mathbf{box} u with v_1, \ldots, v_n for x_1, \ldots, x_n
\]

where v_k is a box-term and \(v_i = v_j with i ≠ j\). There are three cases: \(k = i\Gamma\) \(k = j\Gamma\) for \(k \notin \{i, j\}\). The latter case is easy since the two rules in fact apply in parallel (much like the previous critical pairs). The case \(k \equiv j\) is the same as the case \(k \equiv i\) by permutation of indexes (commutative conversion) and α-renaming. In the last case (\(k \equiv i\)) the term above:

\[
\mathbf{box} u with \ldots, v_i, \ldots, v_j, \ldots for \ldots, x_i, \ldots, x_j, \ldots
\]

where:

\[
v_i = v_j = \mathbf{box} v with w_1, \ldots, w_m for x_1, \ldots, x_m
\]

contracts either to:

\[
\mathbf{box} u[(\mathbf{box} v with z_1, \ldots, z_m for x_1, \ldots, x_j)/x_i]
\]

with \ldots, w_1, \ldots, w_m, \ldots, v_i, \ldots

for \ldots, z_1, \ldots, z_m, \ldots, x_j, \ldots

by (box) or to:

\[
\mathbf{box} u[x_i/x_j] with \ldots, v_i, \ldots for \ldots, x_i, \ldots
\]

by (trace).
The former reduces by (box) \((v_j\) being a box-term) to:

\[
\text{box } u[\text{box } v \text{ with } z_1, \ldots, z_m \text{ for } x_1, \ldots, x_m/ x_i, (\text{box } v \text{ with } z'_1, \ldots, z'_m \text{ for } x_1, \ldots, x_m/ x_j]
\]

with \(u_1, \ldots, u_m, \ldots, u'_1, \ldots, u'_m, \ldots\)

for \(z_1, \ldots, z_m, \ldots, z'_1, \ldots, z'_m, \ldots\)

then by \(m\) applications of (extract) to:

\[
\text{box } u[\text{box } v \text{ with } z_1, \ldots, z_m \text{ for } x_1, \ldots, x_m/ x_i, (\text{box } v \text{ with } z'_1, \ldots, z'_m \text{ for } x_1, \ldots, x_m/ x_j]
\]

with \(u_1, \ldots, u_m, \ldots\)

for \(z_1, \ldots, z_m, \ldots\)

which is also what we get by applying rule (box) to the other term of the critical pair on \(v_i\).

- Between (gc) and itself. If \(i \neq j\) and neither \(x_i\) not \(x_j\) is free in \(u\) then:

\[
\text{box } u \text{ with } \ldots, v_i, \ldots, v_j, \ldots \text{ for } \ldots, x_i, \ldots, x_j, \ldots
\]

reduces in one step either to:

\[
\text{box } u \text{ with } \ldots, v_j, \ldots \text{ for } \ldots, x_j, \ldots
\]

or to:

\[
\text{box } u \text{ with } \ldots, v_i, \ldots \text{ for } \ldots, x_i, \ldots
\]

which both reduce by (gc) to:

\[
\text{box } u \text{ with } \ldots, \ldots, \ldots \text{ for } \ldots, \ldots, \ldots
\]

- Between (gc) and (extract): when \(v_i = v_j\) and \(x_i\) is not free in \(u\) \(\Gamma\) then:

\[
\text{box } u \text{ with } \ldots, v_i, \ldots, v_j, \ldots \text{ for } \ldots, x_i, \ldots, x_j, \ldots
\]

reduces in one (gc) step to:

\[
\text{box } u \text{ with } \ldots, v_j, \ldots \text{ for } \ldots, x_j, \ldots
\]

or to:

\[
\text{box } u[x_i/x_j] \text{ with } \ldots, v_i, \ldots \text{ for } \ldots, x_i, \ldots
\]

Now since \(x_i\) is not free in \(u\) \(\Gamma\) the latter is an \(\alpha\)-renamed version of:

\[
\text{box } u \text{ with } \ldots, v_i, \ldots \text{ for } \ldots, x_j, \ldots
\]

This is just:

\[
\text{box } u \text{ with } \ldots, v_j, \ldots \text{ for } \ldots, x_j, \ldots
\]

since \(v_i = v_j\) \(\Gamma\) so it is \(\sim\) equivalent to:

\[
\text{box } u \text{ with } \ldots, v_j, \ldots \text{ for } \ldots, x_j, \ldots
\]

by the commuting conversion. This closes the confluence diagram.

The case where \(x_j\) instead of \(x_i\) is not free in \(u\) is similar.

- Between (extract) and itself: given \(v_i = v_j\) and \(v_k = v_l\) with \(i \neq j, k \neq l\) and \((i, j) \neq (k, l)\) \(\Gamma\)

\[
\text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n
\]

reduces in one step either to the box of \(u[x_i/x_j]\) with all bindings but the \(j\)th for to the box of \(u[x_k/x_l]\)

with all bindings but the \(i\)th.

If \(j \neq i\) then \(i \neq k\) by assumption \(\Gamma\) and \(v_j = v_l\) so that these two terms are in fact \(\sim\) equivalent.

If \(j \neq i\) then the former reduces by (extract) to the box of \(u[x_i/x_j][x_k/x_l]\) with all bindings but the \(j\)th and the \(i\)th \(\Gamma\) and the latter reduces by (extract) to the box of \(u[x_k/x_l][x_i/x_j]\) with all bindings but the \(j\)th and the \(i\)th. Since \(j \neq i\) these are the same terms.
Theorem 5.7 The $\lambda_{54}$-calculus is confluent.

Proof: Let $\frac{u}{\rightarrow}$ be the relation defined as: $u \frac{1}{\rightarrow} v$ if and only if there is a $\lambda_{54}$-term $u'$ such that $u' \rightarrow^* v$ in $\lambda_{54} \Gamma$ and $E(u') = u$.

We claim that $\frac{1}{\rightarrow}$ is confluent. Indeed let $u \frac{1}{\rightarrow} u_1$ and $u \frac{1}{\rightarrow} u_2$. Then by definition of $\rightarrow$ there are two terms $u_1'$ and $u_2'$ which we obtain from $u$ by adding primes on some $\lambda$-redexes (i.e., such that $E(u_1') = E(u_2') = u$) such that $u_1' \rightarrow^* u_1$ and $u_2' \rightarrow^* u_2$ in $\lambda_{54}$. Build $u'$ from $u$ by adding primes on $\lambda$'s at those positions where $u_1'$ or $u_2'$ has a prime; then again $E(u') = u$ since all these positions are positions of $\lambda$-redexes. By an easy induction on the length of derivations $\Gamma u_1' \rightarrow^* u_1''$ and $u_2' \rightarrow^* u_2''$ in $\lambda_{54} \Gamma$ where $E(u_1'') = u_1$ and $E(u_2'') = u_2$. By Theorem 5.5 $u_1''$ and $u_2''$ have $\lambda_{54}$-normal forms; by Lemma 5.6 these normal forms are the same; call them $v$. Because $v$ is normal, $E(v) = v$ that is $\Gamma v$ is in fact a $\lambda_{54}$-term $\Gamma$ and clearly the reductions from $u_1''$ and $u_2''$ to $v$ translate to $\lambda_{54}$-reductions $\Gamma$ erasing all primes from redexes and replacing $(\beta')\Gamma(\boxx')$ and $(\text{contract'})$-contractions by $(\beta)\Gamma(\boxx)$ and $(\text{contract})$-contractions respectively. Therefore $\frac{1}{\rightarrow}$ and $\frac{1}{\rightarrow}$.

Then we claim that the reflexive transitive closure $\rightarrow^*$ of the one-step $\lambda_{54}$-reduction $\rightarrow$ is equal to the reflexive transitive closure $\frac{1}{\rightarrow}^*$ of $\frac{1}{\rightarrow}$. Indeed $\frac{1}{\rightarrow}$ is a sub-relation of $\frac{1}{\rightarrow};$ and conversely $\frac{1}{\rightarrow}$ is a sub-relation of $\frac{1}{\rightarrow}$; if $u \rightarrow v$ by some rule other than $(\beta)\Gamma$ this is trivial; otherwise just add a prime on the $\lambda$-redex that is contracted and $(\beta')\Gamma$-contract.

Confluence of $\rightarrow^*$ then follows from the confluence of $\frac{1}{\rightarrow}^*$. □

Theorem 5.8 Subject reduction holds in $\lambda_{54}$.

Proof: By case analysis on the reduction rules. The only difficulty is with the $(\boxx)\Gamma(\text{gc})$ and $(\text{contract})$ rules. The result then follows from the $(\beta)$ case from the fact that if $\Gamma, x : \Phi \vdash u : \Phi'$ and $\Gamma \vdash v : \Phi$ are derivable in the system of Figure 4 then $\Gamma \vdash u[v/x] : \Phi'$ is derivable too. This, in turn, follows from an easy induction on $u$ (and the fact that when $v$ is a $\boxx$-term we can only substitute in it with part).

The deep reason why this works is that reduction rules in the typed fragment of $\lambda_{54}$ are really proof transformations $\Gamma$ and that proofness is typeness. Since we have in fact derived the reduction rules from allowed proof transformations $\Gamma$ we won’t bother the reader by translating all this work into yet another language. □

Theorem 5.9 All typed $\lambda_{54}$-terms are strongly normalizing.

Proof: Define the following erasing transformation by structural induction on the terms:

$$D(\boxx \ u \ with \ v_1, \ldots, v_n \ for \ x_1, \ldots, x_n) = D(u)[D(v_1)/x_1, \ldots, D(v_n)/x_n]$$

$$D(\text{unbox} \ u) = D(u)$$

and $D$ does nothing on other constructions — i.e., $\Gamma D(x) = x \Gamma D(\lambda x \cdot u) = \lambda x \cdot D(u) \Gamma D(\text{unbox}) = D(u) \Gamma D(v)$ Notice that this transformation is compatible with the equivalence $\sim$.

Define also the erasing transformation $D(\Phi)$ on formulas $\Phi$ by erasing all boxes $\Box \Gamma$ and similarly on contexts $\Gamma$. An easy structural induction on typing derivations shows that if $\Gamma \vdash u : \Phi$ in $S4\Gamma$ then $D(\Gamma) \vdash D(\Phi)$ in the simply-typed $\lambda$-calculus.

Now let $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_i \rightarrow \ldots$ be a reduction starting from the well-typed $\lambda_{54}$-term $u_1$. Then $D(u_1) \rightarrow D(u_2) \rightarrow \ldots \rightarrow D(u_i) \rightarrow \ldots$ in the simply-typed $\lambda$-calculus $\Gamma$ where $\rightarrow$ is the reflexive closure of reduction in this calculus; indeed every contraction by $(\beta)$ translates by $D$ into a contraction by the same rule $\Gamma$ and if $u_i \rightarrow u_{i+1}$ by some other rule then $D(u_i) = D(u_{i+1})$.

Since the simply-typed $\lambda$-calculus is strongly normalizing there are only finitely many $\rightarrow$ steps that are not equalities in the above reduction. On the other hand those steps that are equalities are erasings of $(\text{unbox})\Gamma(D(\boxx))$ or $(\text{contract})$ steps which are in fact steps in $\lambda_{54}';$ by Theorem 5.5 there can be only finitely many consecutive $\rightarrow$ steps that are equalities. Therefore the original $\lambda_{54}$-reduction is finite. □

24
Definition 5.5 We say that a $\lambda_{S4}$-term is an elimination if and only if it is of the form $vw$ or $\text{unbox } v$.

Let $u$ be a $\lambda_{S4}$-term, and let $(u_i)_{i \geq 0}$ be the sequence of (occurrences of) terms defined as follows: $u_0 = u$, and for each $i \geq 0$, if $u_i$ is an elimination $vw$ or $\text{unbox } v$, then $u_{i+1} = v$; otherwise the sequence stops at index $i$.

When the sequence stops at $n > 0$, we define $d_1(u)$ as $u_{n-1}$, and $d_2(u, y)$ as $u$ where the occurrence of $u_{n-1}$ has been replaced by the variable $y$.

For the following lemma recall that the size of a term is the number of distinct occurrences in it. i.e. the size of a variable is 1, the size of an application $uv$ is the sum of those of $v$ and $u$ plus 1 etc.

Lemma 5.10 For every elimination $u$, $d_1(u)$ is defined.

Moreover, for every variable $y$ that is not free in $u$, $d_2(u, y)$ is defined, has a strictly smaller size as $u$, and $u = d_2(u, y)[d_1(u)/y]$.

Proof: The first claim follows from the fact that the sequence $(u_i)_{i \geq 0}$ always stops (the size of $u_i$ is strictly decreasing) and that $u_1$ is defined since $u$ is an elimination. So it stops at some index $n \geq 1$. Hence $d_1(u)$ is defined.

The second claim follows by an easy induction on the index $n$ at which the sequence $(u_i)_{i \geq 0}$ stops. □

Lemma 5.11 Let $u$ be a well-typed normal elimination. Then $d_1(u)$ is defined, and is of the form $xw$ or $\text{unbox } x$ for some variable $x$.

Proof: As for Lemma 5.11, define the sequence $(u_i)_{i \geq 0}$ as in Definition 5.5 and let $n \geq 1$ be the index at which it stops. Then $u_n$ is not an elimination and the only change is when $u_{n-1}$ is an evaluation $\text{unbox } u_n$.

Then $\Gamma u_n$ cannot be an abstraction. By typing for a quotation box $\_x$ with $\_y$ by normality (rule (unbox) does not apply). So $u_n$ must be a variable $x$. □

Theorem 5.12 Every typable normal $\lambda_{S4}$-term labels the end sequent of some cut-free $LS4$ proof.

Proof: Typable terms are constructions for some LS4 proofs. Then this follows from the same argument as in the proof of Theorem B.5 defining $\pi(u)$ by induction on the size of $u$.

In case $u$ is an elimination the argument is the same as in Theorem B.5 using Lemmas 5.10 and 5.11.

When $u$ is a $\lambda$-abstraction we complete the proofs obtained by induction hypothesis by $(\Rightarrow L)$. The only difficult case is when $u$ is a quotation box $u'$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$, because $u$ is normal, the $x_i$'s are exactly the free variables of $u'$ (rule (ge) does not apply) and no $v_i$ is a quotation (rule (box) does not apply); because $u$ is typable each $v_i$ must be a variable or an elimination.

If some $v_i$ is an elimination then let $u'$ be box $u'$ with $v_1, \ldots, v_{i-1}, d_2(v_i, y), v_{i+1}, \ldots, v_n$ for $x_1, \ldots, x_n$. The size of $u'$ is strictly less than that of $u$. So that $\pi(u')$ is defined by induction hypothesis and we complete it at the bottom by $(\Rightarrow L)$ or $(\Box L)\Gamma s$ in the case of eliminations.

So the only remaining case is when $u$ is box $u'$ with $v_1, \ldots, v_n$ for $x_1, \ldots, x_n$, where the $x_i$'s are the free variables of $u$ and $v_1, \ldots, v_n$ are variables. Because $u$ is normal $\Gamma u$ is not a (extract) redex $\Gamma s$ the variables $v_1 \Gamma \ldots \Gamma v_n$ are pairwise distinct. By induction hypothesis $\Gamma u'[v_1/x_1, \ldots, v_n/x_n]$ (which is strictly smaller than $u$) is a construction for some cut-free proof $\pi(u'[v_1/x_1, \ldots, v_n/x_n])$. Which we complete by adding $(\Box L)$ below $\Gamma$ as follows:

$$\pi(u'[v_1/x_1, \ldots, v_n/x_n])$$

$$(\Box L) \frac{v_1 : \Box \Phi_1, \ldots, v_n : \Box \Phi_n \vdash u'[v_1/x_1, \ldots, v_n/x_n] : \Phi}{v_1 : \Box \Phi_1, \ldots, v_n : \Box \Phi_n \vdash u : \Box \Phi}$$

Indeed because $u$ is typable $\Gamma$ the $v_i$'s must be given boxed types. □

Theorem 5.13 Every typable $\lambda_{S4}$-term has a most general type. Moreover, deciding whether a term is typable and, if so, computing its most general type can be done in polynomial time.
Proof: As in Theorem B.6Γ by a modification of Hindley’s type inferencing algorithm. Typing unboxed v is done as we were typing \( \mathbf{t} \mathbf{u} \mathbf{g} \) and computing \( T(\mathbf{v} \mathbf{u} \mathbf{g}) \) as for \( \mathbf{w} \): compute \( (\mathbf{c} \mathbf{e} \mathbf{s} \mathbf{s} \mathbf{n} \mathbf{s} \mathbf{a} \mathbf{r} \mathbf{o} \mathbf{f} \mathbf{p} \mathbf{o} \mathbf{r} \mathbf{f} \mathbf{s} \mathbf{t} \mathbf{h} \) that are in fact the same, although it is not needed to show that any typable normal term is a construction for some cut-free proof. But rule (\text{ctract}) is needed to this purpose and this forces us to understand the with/or parts as sets of bindings (in fact \( \Gamma \) as maps).

We cannot dispense with rule (\text{ctract}) since without it box \( \lambda u \cdot u z z \) with \( x \) for \( y \) \( z \) would be normal. But then \( \Gamma \) there would be no reasonable cut-free proof corresponding to this term. With rule (\text{ctract}) \( \Gamma \) this rewrites to box \( \lambda u \cdot u z z \) with \( x \) for \( \mathbf{z} \) which corresponds to the following cut-free proof:

\[
\begin{array}{l}
(Ax) \\
(\Rightarrow) \\
(\Rightarrow) \\
(\Rightarrow) \\
(\Gamma) \\
\end{array}
\]

\[
\begin{array}{c}
\boxed{x : \Box \Phi, x : \Psi \vdash x : \Psi} \\
\boxed{x : \Box \Phi, x_1 : \Box \Phi \Rightarrow \Psi \vdash x_1 x : \Psi} \\
\boxed{x : \Box \Phi, x : \Box \Phi \Rightarrow \Psi \vdash \exists x z : x \Rightarrow \Psi} \\
\boxed{x : \Box \Phi \Rightarrow \exists x z : \Box \Phi \Rightarrow \exists x z : \exists x \Rightarrow \Psi} \\
\boxed{\exists x z : \exists x \Rightarrow \Psi} \\
\end{array}
\]

We cannot dispense with the commuting conversion either otherwise we would lose confluence. For example \( \Gamma \) when \( v_2 = v_3 \) is not boxéd \( (u, v) \) instead of the more cumbersome \( \lambda z : z u v \) \( \Gamma \)

\[
\boxed{(x_1, x_3), (x_4, x_5)} \]

rewrites by rule (\text{gc}) to

\[
\boxed{(x_1, x_3), (x_4, x_5)} \]

and by rule (\text{ctract}) to

\[
\boxed{(x_1, x_3), (x_4, x_5)} \]

If \( v_1 \Gamma v_2 \Gamma v_3 \) and \( v_5 \) are distinct variables then these two terms are normal and not \( \alpha \)-convertible and there is no way to close the confluence diagram — except by permuting the second and third bindings.

So it seems that we are forced to have terms in the calculus that behave as sets of \( \Gamma \) lists of bindings. This interpretation is therefore questionable from a computational point of view where unordered structures like finite maps or sets are not considered basic enough. Alternatively the \( \lambda \Sigma \)-calculus is not a computationally adequate formalism for describing how cut-elimination works in \( S \). This is a hint that instead of \textit{explaining} the computation contents of cut-elimination \( \lambda \Sigma \) in fact hides it under the carpet.

Another problem in the \( \lambda \Sigma \)-calculus is related to the rules (\text{gc}) and (\text{ctract}). They really represent particular cases where the implicit weakening and contraction rules of the logic have to be stated explicitly. They do not contribute to the computational meaning of the calculus. From the standpoint of the sequent system we had therefore rather have these two rules changed into equalities \( \Gamma \) that is \( \Gamma \) new clauses in the definition of the commuting conversion relation \( \sim \). On the other hand \( \Gamma \) is more natural to see them as rewrite rules from a computational viewpoint; this is a slight but annoying mismatch. To balance this I’ve define:

**Definition 5.6 (\( \lambda \Sigma \))** The \( \lambda \Sigma \)-calculus is defined as follows.

The \( \lambda \Sigma \)-terms are all equivalence classes of \( \lambda \Sigma \)-terms modulo \( \sim \), where \( \sim \) is the smallest congruence containing \( \sim \), (\text{gc}) and (\text{ctract}). (That is, such that whenever \( u \sim v \), or \( u \rightarrow v \) by the latter rules, then \( u \sim v \).)
The rewrite rules are rules \((\beta), (\text{unbox})\) and \((\text{box})\).

The typing rules are those of Figure 4, plus the following:

\[
\frac{u \approx v, \Gamma \vdash u : \Phi}{\Gamma \vdash v : \Phi}
\]

We need the additional typing rule because rule \((\Pi I)\) is not invariant under \(\approx\). This has the unfortunate consequence that a term might be typable but still have untypable subterms (consider box \(x\) with \(u, v\) for \(x, y\Gamma\) with \(u\) typable but \(v\) untypable). The typed calculus also exhibits non-terminating behaviours: consider for example box \(x\) with \(u, \Omega\) for \(x, y\Gamma\) where \(\Omega\) is an (untypable) term with an infinite rewrite \(\Omega \rightarrow \Omega_1 \rightarrow \Omega_2 \rightarrow \ldots\). Then box \(x\) with \(u, \Omega\) for \(x, y \rightarrow \text{box} x\) with \(u, \Omega_1\) for \(x, y \rightarrow \text{box} x\) with \(u, \Omega_2\) for \(x, y \rightarrow \ldots\) is an infinite rewrite derivation (of \(\approx\)-equivalent terms).

We repair this and at the same time provide a Curry-Howard isomorphism for S4 by considering typed terms that are terms where variables are annotated with their types:\(\text{La} \text{Church:}

**Definition 5.7** The set of typed \(\lambda_{S4}\) pre-terms is described by the same grammar as that of the \(\lambda_{S4}\)-terms, except that each variable has a given type \(\Phi\). If the type of \(x\) is \(\Phi\), we sometimes stress this fact by writing it \(x : \Phi\).

The typing rules of typed pre-terms are given in Figure 4, where contexts are constrained so that in any assumption \(x : \Phi\), \(\Phi\) is the type of \(x\).

The typed \(\lambda_{S4}\)-terms are all typable pre-terms. The \(\lambda_{S4}^n\)-terms are all classes of typed \(\lambda_{S4}\)-terms modulo \(\approx\).

Then every sequence of rewriting steps of typed \(\lambda_{S4}\)-terms terminates but there are still infinite rewriting sequences in \(\lambda_{S4}^n\). A sequence of rewriting steps in \(\lambda_{S4}^n\) is indeed a sequence of terms such that \(u_0 \approx u_0' \rightarrow u_1 \approx u_1' \rightarrow \ldots \rightarrow u_n \approx u_n' \ldots\). We then have for instance the looping rewriting sequence box \(x\) with \(y\) for \(x \approx \text{box} x\) with \(y, (\lambda z : x) u\) for \(x, x' \rightarrow \text{box} x\) with \(y, u\) for \(x, x'\) which can be repeated infinitely many times.

Moreover every \(\lambda_{S4}\)-term (resp. \(\lambda_{S4}^n\)-term) has a unique type. In fact if there is a unique typing derivation for any \(\lambda_{S4}\)-term (provided we agree on the context IT say by taking \(\Gamma\) to be the set of all assumptions \(x : \Phi \Gamma\) where \(\Phi\) is the type of \(x\). This shows that typed \(\lambda_{S4}\)-terms embed in the set of natural deduction proofs in the system of Figure 4.

We still have a dilemma with (gc) and (tract). The natural notion of proof in S4 is given by free-form natural deduction, i.e., proofs as trees of formulas (some of them being discharged assumptions) [GLT89]. In this case it is natural to identify proofs modulo (gc) (we are only interested in the part of the proof that is connected in some way to its conclusion) and modulo (tract) (identical proofs can be used as many times as we wish without having to duplicate them and use them under two different variable names). We can take this into account by considering not equivalent classes modulo \(\approx\) but rather equivalence classes of (gc), (tract)-normal terms modulo \(\approx\). Although this forces a particular rewriting strategy on us it enables us to state a version of the Curry-Howard isomorphism for S4 where typable \(\lambda_{S4}^n\)-terms (equivalently \(\Gamma\) gc, (tract)-normal \(\lambda_{S4}\)-terms) which are unique representatives of \(\lambda_{S4}^n\)-terms are in bijection with free-form natural deduction proofs (equivalently \(\Gamma\) natural deduction proofs in sequent form \(\Gamma\) modulo weakening) \(\Gamma\) and where derivations above the strategy are in bijection with eliminations of detours (of cuts). This leads us to the following definition:

**Definition 5.8 (S4)** The upper (resp. lower) category of derivations \(\overline{S4}\) (resp. \(\underline{S4}\)) is defined as follows.

The objects are all typed \(\lambda_{S4}^n\)-terms. The morphisms are all derivations between \(\lambda_{S4}^n\)-terms (resp. all derivations \(u_0 \approx u_0' \rightarrow u_1 \approx u_1' \rightarrow \ldots \rightarrow u_n \approx u_n'\), where each \(\rightarrow\) step is a \((\beta), (\text{unbox})\) or \((\text{box})\) step, and where for each \(i, 0 \leq i \leq n, u_i'\) is the (gc), (tract)-normal form of \(u_i\)).

The lower category \(\underline{S4}\) is the real category of S4 proofs in the same sense as the category of derivations of the simply-typed \(\lambda\)-calculus is the real category of proofs in minimal logic [GLT89]. On the other hand \(\overline{S4}\) is a more natural category than \(\underline{S4}\).

Both \(\overline{S4}\) and \(\underline{S4}\) have the same problem as the \(\lambda_{S4}\)-calculus: the equivalence relation \(\approx\) is not operational. We shall develop a calculus that is free of this defect in the next sections — although it will have some others.
namely its complexity — giving us new insights into the computational contents of cut-elimination in LS4. Rather surprisingly, it will be quite different from $\lambda_4$: we take it as an indication that a lot was hidden in the rules of $\lambda_4$.

6 Related Works

The search for functional interpretations of modal logics was initiated by Dov Gabbay and Roy de Queiroz [GQ90] who were interested in various non-classical logics. Their interpretation of modal logics (not only S4) is a direct recording of the notion of Kripke models inside the proof structure. (See pp.38–39 G.P. cit.) They introduce world variables $w$, write $\forall w \cdot \Phi(w)$ instead of $\square \Phi$, and provide proof rules that are copied from the usual proof rules of the universal quantifier (we change their notations to match ours) in Hilbert-style: notably, if under the assumption $w : W$ (w is a world in the chosen universe $W$) $\Gamma w$ we can derive $u(w) : \Phi(w) \Gamma$, then we can discharge the assumption $w : W$ and derive $\forall w \cdot u(w) : \square \Phi$. (Discharging the assumption $\forall$ a typical operation of natural deduction is only allowed for world assumptions.)

This attempt brings rather little intuition as to what the proof structure in S4 or any other modal logic really looks like. In particular, we have seen that interpreting a Hilbert-style system for S4 was not particularly easy. Finding the correct reduction rules is trickier than Gabbay and de Queiroz don’t address the problem.

The main point in Curry-Howard interpretations is that the reduction rules should represent a process of cut-elimination in a corresponding formulation of the logic. A much more promising attempt in this direction is due to the linear logic community as far as S4 is concerned. It is indeed remarkable that the rules for propositional linear logic [GLT89] are quite similar to those for S4. Indeed, we rewrite the S4 into the linear logic of formulas, as we have seen in Section 4.3. But the resulting calculus is not quite satisfactory as we have seen in Section 5.2.

Frank Pfenning and Hao-Chi Wong [PW95] have tackled the problem from the opposite viewpoint: namely, they try to understand how S4 works in the hope of understanding more about linear logic. They define a calculus $\lambda^{\Box \overline{\Box}}$ of constructions for S4. They try to solve the problem of the $(\Box R)$ rule by using sequents of the form $\Gamma_1; \Gamma_2; \ldots; \Gamma_n \vdash \Phi$ where $\Gamma_1; \Gamma_2; \ldots; \Gamma_n$ is a context stack, i.e., a list of contexts $\Gamma_i = 1 \leq i \leq n$. This is a good start because the main problem in the $(\Box R)$ rule is that we can make the conclusion from the premisses true without lifting formulas on the left-hand side (in $\Gamma$) which in turn prevents cuts from being pushed upwards. Instead of dropping $\Gamma$ if they push it on the context stack, and provide another rule to get back what they have pushed in a consistent manner. Therefore, they replace the $(\Box R)$ by the following two rules:

\[
\begin{align*}
\Box \text{I} & : \Gamma_1; \ldots; \Gamma_n \vdash \Phi \\
\text{pop} & : \Gamma_1; \ldots; \Gamma_n \vdash u : \Box \Phi \\
\Gamma_1; \ldots; \Gamma_n \vdash u : \Phi & \\
\end{align*}
\]

where $\cdot$ is the empty context $\Gamma$ and where the authors use the word “box” instead of $\Box$ and “unbox” instead of $\cdot$. This system is already closer to the one that we shall develop in Part III in that we can interpret their sequents $\Gamma_1; \ldots; \Gamma_n \vdash u : \Phi$ as the internalized and curried version $\Gamma_1 \vdash \Psi_2 \rightarrow \ldots \rightarrow \Psi_n \rightarrow \Phi$ where for each \( i \in \{1, \ldots, n\} \), $\Psi_i$ is defined as $x_i / x \cdot x \overset{m_i}{\rightarrow} \gamma$ if $\Gamma_1$ is $\Psi_i \rightarrow \Psi_i \rightarrow \gamma$, and conversely $\Gamma$ where we can interpret as sequents with context stacks.

Their system however has several defects. First, although typing is decidable, if it is an NP-complete problem. This is due to the fact that $\text{pop}$ is a parasitic rule which we can guess from the fact that it has no computational effect at all. This was corrected in [DP93] where the authors also note that the box and unbox operators naturally represent an eval/quote-like mechanism.

Second, they define reduction rules that do not cover all cases of cut-elimination: they have to restrict themselves to a reduction strategy that they know is safe. To be harsh, they do not propose...
a calculus but only a safe strategy. Defining such a safe strategy (for cut-elimination) is easy in LS4: just delay the elimination of cuts between (□R) on a formula on its left-hand side and some other subproof until this subproof is cut-free. Formulating this strategy is trickier at the level of λ-terms and it seems to have been their motive. The λS4-calculus as well as the λewQ-calculus of Part II does not have to be restricted to special strategies.

As a final reference let's mention the work done in the Scheme community to make better macro packages. As Davies and Pfenning have shown [DP95] `quote` delays some computation and `eval` requests the value of the delayed computation. Macros are just programs that should be evaluated by the compiler front-end to produce new programs. There is a similarity there with programs which would compute at level 0 and return a piece of code at level 1 to interpret later on. It is no surprise that early macro packages in Lisp were implemented using `eval` in the macro-expander and macros were basically returned quoted programs. It is therefore no surprise either that proper scoping rules for so-called hygienic macro systems had to be precisely stated. One of the main results in this area is the invention of syntactic closures [BR88] to solve the scoping problem. Let us just note that box terms in λS4 are precisely that: closures.

Acknowledgements

Many thanks to Ian Mackie and to Anselm Baird-Smith for many fruitful discussions, to Pierre Boury for his early help, to Eric Goubault for discussions on relations between logic and category theory, to Alain Deutsch for pointing out to me Frank Pfenning and Hao-Chi Wong’s paper, to Gilles Dowek for telling me about explicit substitutions and to Thérèse Hardin for discussing about them.
A Equivalence of Hilbert and Gentzen Styles

Let's write $\Gamma \vdash \Phi$ if $\Phi$ can be deduced from the set of assumptions $\Gamma$ in the Hilbert-style system $\Gamma$ and $\Gamma \vdash \Phi$ if the sequent $\Gamma \vdash \Phi$ is deductible in the Gentzen-style sequent system.

We prove the following:

**Theorem A.1** $\Gamma \vdash \Phi$ if and only if $\Box \Gamma \vdash \Phi$.

where $\Box \Gamma$ is the result of boxing all formulas in $\Gamma$.

A.1 $\vdash$ Implies $\vdash$

We show how to transform a Hilbert-style proof of $\Gamma \vdash \Phi$ into a sequent proof of $\Box \Gamma \vdash \Phi$. Instead of just producing a proof, we shall leave the constructions with as to help visualize the kind of terms that they produce.

The construction is by induction on the length of the Hilbert-style proof. The base cases are:

- if $\Phi$ is a member of $\Gamma$ then we translate this to the following proof:
  
  \[
  \begin{array}{c}
  (Ax) \\
  \Gamma, x : x \vdash \Phi
  \end{array}
  \]

- if $\Phi$ is an instance of one of the axiom schemas:

  (s) then we produce the following proof:

  \[
  \begin{array}{c}
  (Ax) \\
  \Gamma, x : \Phi \vdash \Phi \\
  \Gamma, x : \Phi \vdash \Phi
  \end{array}
  \]

- $\Rightarrow$ then we build:

  \[
  \begin{array}{c}
  (Ax) \\
  \Gamma, x : \Phi \vdash \Phi
  \end{array}
  \]

- $\Rightarrow$ then we build:

  \[
  \begin{array}{c}
  (Ax) \\
  \Gamma, x : \Phi \vdash \Phi
  \end{array}
  \]

- $\Rightarrow$ then we build:

  \[
  \begin{array}{c}
  (Ax) \\
  \Gamma, x : \Phi \vdash \Phi
  \end{array}
  \]

where $\Gamma'$ is any context for example $\Box \Gamma$.

(k) then we build:

\[
\begin{array}{c}
(Ax) \\
\Gamma', x : \Phi \vdash \Phi
\end{array}
\]

where $\Gamma' = \Box \Gamma$.

(K) (now we come to the modal part):

\[
\begin{array}{c}
(Ax) \\
\Gamma', x : \Phi \vdash \Phi
\end{array}
\]
Notice that it is now essential that $\Box \Gamma$ be boxed so that we can apply rule $(\Box R)$.

\begin{align*}
(A) & \quad \Gamma, x : \Box \Phi \vdash x : \Phi \\
(\Box I) & \quad \Gamma' \vdash \Box \Phi \vdash \Box \Phi \vdash x : \Phi \\
(R) & \quad \Gamma' \vdash x : \Box \Phi \vdash \Box \Phi \Rightarrow \Phi
\end{align*}

where $\Gamma'$ is $\Box \Gamma$.

(4) Produce:

\begin{align*}
(A) & \quad \Gamma, x : \Box \Phi \vdash x : \Box \Phi \\
(\Box R) & \quad \Gamma, x : \Box \Phi \vdash x' : \Box \Box \Phi \\
(R) & \quad \Gamma \vdash \lambda x \cdot x' : \Box \Box \Phi \Rightarrow \Box \Phi
\end{align*}

Notice again that it is essential that $\Box \Gamma$ be boxed so that we can apply rule $(\Box R)$.

Now assume that we have a Hilbert-style proof of $\Gamma \vdash \Phi$. Then we prove that the same happens with $\Gamma \vdash \Phi$:

- If $\Phi$ was obtained by modus ponens (rule (MP)), i.e., we have strictly shorter proofs of $\Gamma \vdash \Phi_1 \Rightarrow \Phi$ and of $\Gamma \vdash \Phi_1$ then we get derivations $(\pi)$ of $\Gamma' \vdash \Phi_1 \Rightarrow \Phi$ and $(\pi')$ of $\Gamma' \vdash \Phi_1$ where $\Gamma' = \Box \Gamma$.

Therefore the following:

\begin{align*}
(\pi) & \quad \vdash x_2 : \Phi_1 \\
(\pi') & \quad \vdash x_3 : \Phi_1 \\
(\Box R) & \quad \vdash x_2 : \Phi_1 \vdash x_3 : \Phi_1
\end{align*}

is a proof of $\Gamma' \vdash \Phi$. Notice that this is the only case where we need to use the $(\Box R)$ rule.

- If $\Phi$ is obtained by the necessitation rule (Nec) then $\Phi$ is of the form $\Box \Phi' \Gamma$ where $\Gamma \vdash \Phi'$ has a strictly shorter proof. The latter translates to a derivation $(\pi)$ of $\Box \Gamma \vdash \Phi'$, then:

\begin{align*}
(\pi) & \quad \vdash x_2 : \Phi_1 \\
(\Box R) & \quad \vdash \Box \Gamma \vdash \Box \Phi' \Rightarrow \Box \Phi'
\end{align*}

is a derivation of $\Box \Gamma \vdash \Phi$. Notice that we need $\Box \Gamma$ to be boxed so that $(\Box R)$ is applicable.

QED.

A.2 $\vdash$ implies $\Rightarrow$

We now show that if $\Box \Gamma \vdash \Phi$ is derivable in the sequent system then $\Gamma \vdash \Phi$ in the Hilbert-style deduction system.

The main difficulty in translating proofs to Hilbert style is the translation of the $(\Rightarrow R)$ rule. Indeed we need to transform proofs of sequents of the form $\Gamma, \Phi_1 \vdash \Phi_2$ into proofs of $\Gamma \vdash \Box \Phi_1 \Rightarrow \Phi_2$.

Lemma A.2 In the non-modal fragment (s), (k), $\Gamma, \Phi_1 \vdash \Phi_2$ is provable if and only if $\Gamma \vdash \Phi_1 \Rightarrow \Phi_2$ is provable.

Proof: If $\Phi_1 \Rightarrow \Phi_2$ is provable from $\Gamma$ then $\Phi_1 \Rightarrow \Phi_2$ is also provable from $\Gamma, \Phi_1$. Moreover, $\Phi_1$ is provable from $\Gamma, \Phi_1 \Rightarrow \Phi_2$ and by modus ponens $\Phi_2$ is provable from $\Gamma, \Phi_1$.

Conversely assume that $\Phi_2$ is provable from $\Gamma, \Phi_1$. We build a proof of $\Gamma \vdash \Phi_1 \Rightarrow \Phi_2$ by induction on the length of the proof of $\Gamma, \Phi_1 \vdash \Phi_2$.

31
• if $\Phi_2$ is a formula in $\Gamma$ then we can first prove $(\Phi_1 \Rightarrow (\Phi_2 \Rightarrow \Phi_1) \Rightarrow \Phi_1) \Rightarrow (\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_1) \Rightarrow (\Phi_2 \Rightarrow \Phi_1)$ by instantiating $(s)$ (replace $\Phi_2$ by $\Phi_1 \Gamma$ and $\Phi_2$ by $\Phi_1$); then we can prove $(\Phi_1 \Rightarrow (\Phi_2 \Rightarrow \Phi_1) \Rightarrow \Phi_1)$ by instantiating $(k)$ (replace $\Phi_2 \Rightarrow \Phi_1$). By modus ponens we infer $(\Phi_1 \Rightarrow \Phi_1 \Rightarrow (\Phi_1 \Rightarrow \Phi_1))$. But then $(\Phi_1 \Rightarrow \Phi_1)$ is another instance of $(k)$ so by modus ponens again we infer $\Phi_1 \Rightarrow \Phi_1 \Gamma$. This is a proof of $\Phi_1 \Rightarrow \Phi_2$. (Notice the analogy with the translation from the $\lambda$-term $\lambda x \cdot x$ to the combinatory term $SKK$.)

• if $\Phi_2$ is a formula in $\Gamma$ then we can first prove $\Phi_2 \Rightarrow \Phi_1 \Rightarrow \Phi_2 \Gamma$ which is an instance of $(k) \Gamma \Gamma$ then apply modus ponens on this and $\Phi_2$ to get $\Phi_1 \Rightarrow \Phi_2$. (Notice the analogy with the translation from $\lambda x \cdot y$ to $K y$.)

• if $\Phi_2$ was deduced by modus ponens i.e., there is a formula $\Phi_3$ which is provable from $\Gamma$, $\Phi_1 \Gamma\delta$ and such that $\Phi_3 \Rightarrow \Phi_2$ is provable from $\Gamma$, $\Phi_1$. Then by induction hypothesis $\Gamma \Gamma \Rightarrow \Phi_2 \Rightarrow \Phi_3$ and $\Phi_3 \Rightarrow \Phi_2$ are provable from $\Gamma$. Then we produce $(\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3) \Rightarrow (\Phi_1 \Rightarrow \Phi_3)$ which is an instance of $(s)$ and apply modus ponens twice to get $\Phi_1 \Rightarrow \Phi_2$. (Notice the analogy with the translation from $\lambda x \cdot t \ell^t \Gamma$ to $S(\lambda x \cdot \ell)(\lambda x \cdot \ell)$.)

• if $\Phi_2$ was an instance of $(s)$ or $(k) \Gamma \Gamma$ then we produce $\Phi_2 \Rightarrow \Phi_1 \Rightarrow \Phi_2 \Gamma$ which is an instance of $(k) \Gamma \Gamma$ then apply modus ponens to get $\Phi_1 \Rightarrow \Phi_2$.

□

Lemma A.3 $\Gamma, \Phi_1, \Phi_2$ if and only if $\Gamma \vdash \Box \Phi_1 \Rightarrow \Phi_2$.

Proof: If $\Box \Phi_1 \Rightarrow \Phi_2$ is provable from $\Gamma$ then $\Box \Phi_1 \Rightarrow \Phi_2$ is also provable from $\Gamma$, $\Phi_1$. Moreover, if $\Phi_1$ is provable from $\Gamma$, $\Phi_1$ by necessitation then by modus ponens $\Phi_2$ is provable from $\Gamma$, $\Phi_1$.

Conversely, if $\Phi_2$ is provable from $\Gamma$, $\Phi_1$. We build a proof of $\Gamma \vdash \Box \Phi_1 \Rightarrow \Phi_2$ by induction on the length of the proof of $\Gamma$, $\Phi_1 \Rightarrow \Phi_2$. As in the proof of Lemma A.2, in all cases of the latter lemma if we assume that $\Gamma, \Box \Phi_1 \Rightarrow \Phi_2$ is also provable by using $(T)$ and modus ponens $\Gamma \Gamma$ then we replace all uses of $\Phi_1$ by $\Box \Phi_1$ in the latter proof. This handles the case where $\Phi_2$ was obtained by any rule other than necessitation or by instantiating the non-modal axioms.

If $\Phi_2$ was obtained by necessitation $\Gamma$ i.e., $\Phi_2$ is $\Box \Phi_3$ where $\Gamma, \Phi_1 \Rightarrow \Phi_3$ is provable then by induction hypothesis we can deduce $\Box \Phi_1 \Rightarrow \Phi_3$ from $\Gamma$. Then by necessitation we infer $\Box \Box \Phi_1 \Rightarrow \Phi_3$. Then we produce $\Box (\Box \Phi_1 \Rightarrow \Phi_3) \Rightarrow \Box \Box \Phi_1 \Rightarrow \Box \Phi_2$ which is an instance of $(K)$. By modus ponens we get $\Box \Box \Phi_1 \Rightarrow \phi_3$. Produce $\Box \Box \Phi_1 \Rightarrow \Box \Phi_3 \Rightarrow \Box \Phi_1 \Rightarrow (\Box \Phi_1 \Rightarrow \Box \Phi_3)$ which is an instance of $(\Gamma) \Gamma \Gamma$ infer $\Box \Phi_1 \Rightarrow \Box \Phi_3$ by modus ponens. Now produce $(\Box \Phi_1 \Rightarrow \Box \Phi_1 \Rightarrow \Box \Phi_3) \Rightarrow (\Box \Phi_1 \Rightarrow \Box \Phi_1) \Rightarrow (\Box \Phi_1 \Rightarrow \Box \Phi_3)$ by modus ponens. But $\Box \Phi_1 \Rightarrow \Box \Phi_3$ is an instance of $(\Gamma) \Gamma \Gamma$ from which we deduce $\Box \Phi_1 \Rightarrow \Box \Phi_2$ i.e., $\Box \Phi_1 \Rightarrow \Phi_2$ by modus ponens.

If $\Phi_2$ was an instance of $(K) \Gamma \Gamma(T)$ for $(A)$ then produce $\Phi_2 \Rightarrow \Box \Phi_1 \Rightarrow \Phi_2$ which is an instance of $(K) \Gamma \Gamma$ and apply modus ponens to get a proof of $\Box \Phi_1 \Rightarrow \Phi_2$.

□

Lemma A.4 Assume that $\Gamma \vdash \Phi_1 \Rightarrow \Phi_2$ and $\Gamma \vdash \Phi_2 \Rightarrow \Phi_3$. Then $\Gamma \vdash \Phi_1 \Rightarrow \Phi_3$.

Proof: By modus ponens between $(k)$ and $\Phi_2 \Rightarrow \Phi_3$ we get $\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3$. Then by modus ponens with $(s)$ we get $(\Phi_1 \Rightarrow \Phi_2) \Rightarrow (\Phi_1 \Rightarrow \Phi_3)$. Then modus ponens with $\Phi_1 \Rightarrow \Phi_2$ proves $\Phi_1 \Rightarrow \Phi_3$. (Notice how similar this is to the construction of the combinator $B$ such that $Bx y z = x(yz)$ as $Bx y = S(Kx y)$.)

□

Lemma A.5 If $\Gamma \vdash \Phi$, then $\Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_N \Rightarrow \Phi$, for any formulas $\Phi_1, \Phi_2, \ldots, \Phi_N$.

Proof: If $n = 0$ this is trivial. Now assume the property true for $n \Box \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi$ is provable from $\Gamma \Gamma$ and prove it for $n + 1$: produce $(\Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi) \Rightarrow \Phi_1 \Rightarrow (\Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi) \Gamma \delta$ which is an instance of $(k) \Gamma \Gamma$ by modus ponens $\Gamma \Gamma$ infer $\Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi$. (Notice that this is the same as translating the $\lambda$-term $\lambda x_1 \cdot \lambda x_2 \cdot \ldots \cdot \lambda x_n \cdot \Gamma$ where $x$ is not one of the $x_i$s into the combinatory term $K(K(\ldots K x \ldots) \ldots)$.) □

Lemma A.6 For any formulas $\Phi_1, \Phi_2, \ldots, \Phi_n$, and $1 \leq i \leq n$, we have $\Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_i$. 32
**Proof:** Without loss of generality we can assume \( i = 1 \); if \( i > 1 \) we construct a proof of \( \Gamma \vdash \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_1 \) from which we deduce a proof of \( \Gamma \vdash \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_1 \) by Lemma A.5.

If \( n = 1 \) then \( \Phi_2 \vdash \Phi_n \vdash \Phi_1 \) is provable by modus ponens with \((s)\Gamma(k)\) and \((k)\) as in the first case of the proof of Lemma A.2. Then by Lemma A.5 \( \Gamma \vdash \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_1 \).

Assume that \( \Gamma \vdash \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_1 \) is provable from \( \Gamma \) for all formulas \( \Phi_1 \Gamma \Phi_2 \Gamma \ldots \Gamma \Phi_n \); we show that this holds still when \( n \) is replaced by \( n + 1 \). So take \( n + 1 \) formulas \( \Phi_1 \Gamma \Phi_2 \Gamma \ldots \Gamma \Phi_{n+1} \). Then \( \Phi_1 \Rightarrow \Phi_3 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi_1 \) is provable. Let’s name \( \Psi \) the formula \( \Phi_3 \Rightarrow \Phi_4 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi_1 \). So that \( \Phi_1 \vdash \Psi \) is provable.

We produce \((\Psi \Rightarrow \Phi_2 \Rightarrow \Psi) \Rightarrow \Phi_1 \Rightarrow (\Psi \Rightarrow \Phi_2 \Rightarrow \Psi) \Gamma \) which is an instance of \((k)\Gamma(k)\) and \( \Psi \Rightarrow \Phi_2 \Rightarrow \Psi \Gamma \) which is another instance of \((k)\) applying modus ponens on these two \( \Psi \Gamma \) infer \( \Phi_1 \Rightarrow \Psi \Rightarrow \Phi_2 \Rightarrow \Psi \). Now \( \Psi \) we produce \((\Phi_1 \Rightarrow \Psi \Rightarrow \Phi_2 \Rightarrow \Psi) \Rightarrow (\Phi_1 \Rightarrow \Psi) \Rightarrow (\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Psi) \Gamma \) which is an instance of \((s) \Gamma \) and we apply modus ponens with the latter formula to get \( \Phi_1 \Rightarrow \Psi \Rightarrow (\Phi_1 \Rightarrow \Phi_2) \Rightarrow \Psi \). Now \( \Psi \) we can apply modus ponens on the formula \( \Phi_1 \Rightarrow \Psi \Gamma \) which was assumed provable at the beginning \( \Gamma \) and get \( \Phi_1 \Rightarrow \Psi \Gamma \). Let \( \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi_1 \). (Notice that this corresponds to the translation of \( \lambda x_1 . \lambda x_2 . \ldots \lambda x_n . x_1 \) into \( \text{S}(K K)(\ldots (S(K K)(K K) \ldots )) \).)

\( n \) \hspace{1cm} \( \ldots \)

\( \square \)

**Lemma A.7** If \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi \Rightarrow \Psi' \) and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi' \), then \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi \Rightarrow \Psi' \).

**Proof:** By induction on \( n \). If \( n = 0 \) this is trivial. Otherwise assume this works for all sequences \( \Phi_1 \Gamma \Phi_2 \Gamma \ldots \Gamma \Phi_n \) of \( n \) formulas and show it works for \( n + 1 \). Assume that \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_{n+1} \Rightarrow \Psi \Rightarrow \Psi' \) and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_{n+1} \Rightarrow \Psi \). But \( \Phi_{n+1} \Rightarrow \Psi \Rightarrow \Psi' \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Gamma \) is an instance of \((s) \Gamma \) so by Lemma A.5 \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Gamma \). Then \( \Gamma \) apply the induction hypothesis to this and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Gamma \) to get \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \). Apply the induction hypothesis again to this and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_{n+1} \Rightarrow \Psi \Gamma \) to get \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow (\Phi_{n+1} \Rightarrow \Psi) \Gamma \) which is the desired conclusion. \( \square \)

Lemma A.7 says that we have modus ponens under a prefix \( \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \). By Lemma A.5 we also have all tautologies under this prefix (those deducible from \((s) \) and \((k) \)).

**Lemma A.8** Assume that \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi \) and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi_2 \). Then \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi_2 \Rightarrow \Psi_3 \).

**Proof:** Same proof as Lemma A.4 using Lemma A.7 instead of modus ponens. \( \square \)

We now have all we need to prove that if \( \square \Gamma \vdash \Phi \) is derivable in the sequent system \( \Gamma \) then \( \Phi \) is provable from \( \Gamma \) in the Hilbert-style system. More generally we prove that if \( \square \Gamma, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi \) is derivable then \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi \) holds in the Hilbert-style system. We do this by induction on the number of steps in a sequent derivation of \( \square \Gamma, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi \).

\((Ax)\) If \( \Box \Gamma, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi \) is derived by \((Ax)\) \( \Gamma \) then either \( \Phi = \Phi_1 \) for some \( i \) and we conclude by Lemma A.6, or \( \Gamma \) has the form \( \Gamma' \), \( \Box \Gamma' \) and \( \Phi = \Box \Phi' \); a derivation of \( \Box \Gamma' \), \( \Box \Phi' \) by \((Ax)\) yields a proof of \( \Gamma' \), \( \Box \Phi' \) by necessitation on the assumption \( \Phi \Gamma \) and then a proof of \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi \) by Lemma A.5.

\((\Rightarrow L)\) If \( \Box \Gamma, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi \) is derived by \((\Rightarrow L)\) \( \Gamma \) then some \( \Phi_i \) has the form \( \Psi \Rightarrow \Psi' \) (without loss of generality let it be \( \Phi_n \)) and by induction hypothesis \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Psi \Rightarrow \Phi \) and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Psi \). By using \((k) \Gamma \) Lemma A.5 and Lemma A.7 we transform the latter two into \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Phi \Rightarrow \Psi \Rightarrow \Phi \) (statement 1) and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Phi \Rightarrow \Psi \) (statement 2) respectively. By applying Lemma A.7 on statement 2 and \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Phi \Rightarrow \Phi \Rightarrow \Psi \) (an instance of Lemma A.6) since \( \Phi_n \Rightarrow \Psi \Rightarrow \Psi' \) we get \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Phi \Rightarrow \Psi \). Reapplying Lemma A.7 between the latter and the former \( \Gamma \) we get \( \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_{n-1} \Rightarrow \Phi \Rightarrow \Phi \).

\((\Rightarrow R)\) If \( \Box \Gamma, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi \) is derived by \((\Rightarrow R)\) \( \Gamma \) then \( \Phi \Rightarrow \Psi \Rightarrow \Psi \Gamma \) and \( \Box \Gamma, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Psi \Rightarrow \Psi' \) is derivable. By induction hypothesis \( \Box \Gamma \vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi \Rightarrow \Psi' \Gamma \).
(Cut) If □Γ, Φ₁, Φ₂, ..., Φₙ ⊢ Φ is derived by (Cut) then by induction hypothesis Γ ⊢ Φ₁ → Φ₂ → ... → Φₙ → Ψ and Γ ⊢ Φ₁ ⊢ Φ₂ ⊢ ... ⊢ Φₙ ⊢ Ψ ⊢ Φ. For some formula Φ. We then conclude by applying Lemma A.7.

(□L) If □Γ, Φ₁, Φ₂, ..., Φₙ ⊢ Φ is derived by (□L) then we can assume that Γ = Γ', Φ₀ Γ' and that by induction hypothesis Γ' ⊢ Φ₁ ⊢ Φ₂ → ... → Φₙ → Φ. In particular Γ ⊢ Φ₀ Γ ⊢ Φ₁ ⊢ Φ₂ → ... → Φₙ → Φ. But Γ ⊢ Φ₀ by assumption so by modus ponens Γ ⊢ Φ₁ ⊢ Φ₂ → ... → Φₙ → Φ.

(□R) If □Γ, Φ₁, Φ₂, ..., Φₙ ⊢ Φ is derived by (□R) then Φ is □Ψ Γ and □Γ, Γ' ⊢ Ψ is derivable where Γ' is the subset Φ₁', ..., Φₙ', Γm ≤ nΓ of formulas in Φ₁, ..., Φₙ that are boxed. The (□R) rule then infers □Γ, Γ', Γ ⊢ Φ Γ where Γ' is the set Φ₀', ..., Φₙ'-m' of non boxed formulas in Φ₁, ..., Φₙ. By induction hypothesis Γ ⊢ Φ₁' → ... → Φₙ'-m' → Ψ is provable and we want to show that Γ ⊢ Φ₁ ⊢ Φ₂ → ... → Φₙ → □Ψ is provable i.e. Γ ⊢ Φ₁' → ... → Φₙ'-m' → Φ₁' → ... → Φₙ' → □Ψ is provable. First we prove by necessitation Γ ⊢ □(Φ₁' → ... → Φₙ'-m' → Ψ) then we use modus ponens with (K) m times to prove Γ ⊢ □Φ₁' → ... → □Φₙ'-m' → □Ψ. We now prove that Γ ⊢ Φ₁' → ... → Φₙ'-m' → □Φ₁' → ... → □Φₙ'-m' → □Ψ by induction on i. If i = 0 then this is the result above. So assume that Γ ⊢ Φ₁' → ... → Φₙ'-m' → □Φ₁' → ... → □Φₙ'-m' → □Ψ is provable. Then Γ Φ₁' → □Φ₁' is provable by (4) because Φ₁' is boxed. By Lemma A.8 we get Γ ⊢ Φ₁' → Φ₁'-1 → Φ₁' → □Φ₁'-1 → ... → □Φₙ'-m' → □Ψ.

So now in particular we can prove Γ ⊢ Φ₁' → ... → Φₙ'-m' → □Ψ Γ and by Lemma A.5 the desired result Γ ⊢ Φ₁' → ... → Φₙ'-m' → Φ₁' → ... → Φₙ' → □Ψ.

QED.
B Properties of the $\lambda$-Calculus

**Theorem B.1** The $\lambda$-calculus is confluent.

**Proof:** Consider the reduction $R$ defined by all rules but $(\beta)$. $(\beta)$ is itself confluent.

$R$ is defined by a first-order term rewriting system that has no non-trivial critical pairs Hence $R$ is locally confluent. Moreover the size of each left-hand side is strictly greater than the right-hand side so that $R$ terminates. Therefore $R$ is confluent.

We now show that the reflexive transitive closures $\beta^*$ and $R^*$ of $(\beta)$ and $R$ commute which will entail the confluence of the calculus by Hindley-Rosen’s Lemma (Proposition 3.3.5 in [Bar84]). Consider three terms $u \Gamma v \Gamma v'$ such that $u \beta^* v$ and $u \beta^* v'$. The redex $r$ that is contracted to get $v'$ disjoin the redex $r'$ which is contracted to get $v$: therefore we get the same term $w$ by contracting the residual of $r$ in $v'$ (if there is still one otherwise let $w$ be $v'$) or contracting all residuals of $r'$ in $v$. So $w$ reduces to $u \Gamma w$ and $v'$ reduces in at most one step to $w$. So $\beta^*$ and $R^*$ (this is proposition 3.3.6 in [Bar84]).

**Theorem B.2** All typed $\lambda$-terms are strongly normalizing.

**Proof:** We shall use the fact that simply-typed $\lambda$-terms are strongly normalizing and rewrite $\lambda$-terms of the form $u \Gamma w$ as $u \Gamma (\lambda x \cdot x \cdot u)$. Typable $\lambda$-terms of some type then translate to typable $\lambda$-terms of the same type where $\square$’ symbols have been erased.

Let’s write $u$ the translation of $u$. Given a reduction sequence $u_1 \to u_2 \to \ldots \to u_n \to \ldots$ we interpret reduction steps $u_i \to u_{i+1}$ that rewrite a $\lambda$-redex as the same reduction steps again $u_i \to u_{i+1}$; and we rewrite reduction steps (where $\rho(u)$ rewrites to $u$) as two $\beta$-reduction steps $(\lambda x \cdot x)(\lambda x \cdot x)u \to (\lambda x \cdot x)u \to u$. As any typable term in the source language translates to a typable term the translated sequence terminates [Bar84]. But the length of the translated sequence is at least that of the original. Hence it terminates $\Gamma$ too. □

**Definition B.1** We say that a $\lambda$-term is an elimination if and only if it is of the form $uv$ or $v$. Let $u$ be a $\lambda$-term, and let $\langle u_i \rangle_{i \geq 0}$ be the sequence of (occurrences of) terms defined as follows: $u_0 = u$, and for each $i \geq 0$, if $u_i$ is an elimination $uv$ or $v$, then $u_{i+1} = v$; otherwise the sequence stops at index $i$.

When the sequence stops at $n > 0$, we define $d_1(u)$ as $u_{n-1}$, and $d_2(u, y)$ as $u$ where the occurrence of $u_{n-1}$ has been replaced by the variable $y$.

For the following lemmas recall that the size of a term is the number of distinct occurrences in it. I.e. the size of a variable is $1$ the size of an application $uv$ is the sum of those of $v$ and $u$ plus $1$ etc.

**Lemma B.3** For every elimination $u$, $d_1(u)$ is defined.

Moreover, for every variable $y$ that is not free in $u$, $d_2(u, y)$ is defined, has a strictly smaller size as $u$, and $u = d_2(u, y)[d_1(u)/y]$.

**Proof:** The first claim follows from the fact that the sequence $\langle u_i \rangle_{i \geq 0}$ always stops (the size of $u_i$ is strictly decreasing) and that $u_i$ is defined since $u$ is an elimination. So it stops at some index $n \geq 1$; hence $d_1(u)$ is defined.

The second claim follows by an easy induction on the index $n$ at which the sequence $\langle u_i \rangle_{i \geq 0}$ stops. □

**Lemma B.4** Let $u$ be a well-typed normal elimination. Then $d_1(u)$ is defined, and is of the form $x\Gamma w$ or $\cdot x$ for some variable $x$.

**Proof:** Define the sequence $\langle u_i \rangle_{i \geq 0}$ as in Definition B.11 and let $n \geq 1$ be the index at which it stops. Then $u_{n-1}$ is not an elimination.

If $u_{n-1}$ is an application $u_{n-1} \Gamma v_{n-1}$ cannot be a quotation $\ast v$, because $u_{n-1}$ is well-typed; it cannot be a $\lambda$-abstraction because $u_{n-1}$ is normal. So $u_{n-1}$ must be a variable $x$ and $d_1(u) = u_{n-1} = xw$.

If $u_{n-1}$ is an evaluation $u_0 \Gamma$ then $u_0$ cannot be an abstraction by typing so a quotation by normality. So in both cases $u_{n-1}$ must be a variable $x$.

Finally $u_{n-1}$ is always an elimination by definition of the sequence $\Gamma$ so there are no other cases. □

**Theorem B.5** Every typable normal $\lambda$-term labels the end sequent of some cut-free $\text{LS}_4$ proof.
Proof: We build this cut-free proof \( \pi(u) \) by induction on the size of the normal term \( u \) using the previous Lemmas.

If \( u \) is a variable \( x \) then \( \pi(x) \) is an instance of \( (Ax) \).

If \( u \) is an abstraction \( \lambda x \cdot \Gamma \) then \( \pi(u) \) is \( \pi(v) \) completed with an instance of \( (\Rightarrow R) \) at the bottom. The process is similar if \( u \) is a quotation \( (\sqcap R) \).

If \( u \) is an elimination \( \text{then choose a fresh variable } y \). By Lemma B.3 \( d_1(u) \) and \( d_2(u, y) \) are defined and \( u = d_2(u, y)[d_1(u)/y] \). By Lemma B.4 we have two subcases:

- \( d_1(u) = xv \) for some variable \( x \) and some well-typed normal term \( v \). Recall that the size of \( d_2(u, y) \) is strictly less than that of \( u \); also \( v \) is strictly smaller than \( u \) if \( u \) is a type. So we can apply the induction hypothesis and build \( \pi(u) \) as:

\[
\begin{align*}
\pi(d_2(u, y)) & \quad \pi(v) \\
\vdots & \quad \vdots \\
\Gamma, y : \Phi' & \vdash d_2(u, y) : \Phi \\
(\Rightarrow L) & \\
\Gamma, x : \Phi'' & \vdash d_2(u, y)[xv/y] : \Phi'
\end{align*}
\]

- or \( d_1(u) = x \) for some variable \( x \). Then define \( \pi(u) \) as:

\[
\begin{align*}
\pi(d_2(u, y)) & \\
\vdots & \\
\Gamma, y : \Phi' & \vdash d_2(u, y) : \Phi \\
(\Box L) & \\
\Gamma, x : \Box \Phi & \vdash d_2(u, y)[xv/y] : \Phi
\end{align*}
\]

\( \Box \)

**Theorem B.6** Every typable \( \lambda \)-term has a most general type. Moreover, deciding whether a term is typable and, if so, computing its most general type can be done in polynomial time.

Proof: By a slight modification of Hindley’s type reconstruction algorithm [Hin69]. (This is the basis of Milner’s algorithm for ML [Mil78] without the notion of generalization of type variables.)

We extend the algebra of formulas (types) with type meta-variables \( \Gamma, \beta \Gamma, \ldots \Gamma \) and consider substitutions \( \sigma \) from type meta-variables to types. For any type \( \Phi \Gamma \Phi \sigma \) denotes the result of applying the substitution \( \sigma \) to \( \Phi \). We assume that if \( \Gamma \) is a context \( x_1 : \Phi_1, \ldots, x_m : \Phi_m \Gamma \) then \( \Gamma \sigma \) denotes the context \( x_1 : \Phi_1 \sigma, \ldots, x_m : \Phi_m \sigma \). \( \sigma \Phi \) denotes the composition of \( \sigma \) and \( \Phi \) defined as the substitution such that \( \Phi(\sigma \Phi) = (\Phi \sigma) \sigma' \) for any \( \Phi \). A substitution or type \( T \) is more general than \( T' \) if and only if \( T' \) can be expressed as \( T \sigma' \) for some substitution \( \sigma' \). We extend the notion to couples of such objects in the obvious way.

We define a type inference algorithm \( T(\Gamma, u) \) that returns a most general couple \( (\sigma, \Phi) \) such that \( \Gamma \sigma \vdash u : \Phi \) is derivable for fails if there is no couple \( (\sigma, \Phi) \) making \( \Gamma \sigma \vdash u : \Phi \) derivable. It works by reconstructing a natural deduction proof from the bottom up by structural induction on \( u \) guessing the formulas in the premises that do not appear in the conclusion of the rules by generating fresh type variables \( \Gamma \) and solving equality constraints between types by unification.

To infer the most general substitution and type of \( u \) under \( \Gamma \) we compute \( (\sigma, \Phi) = T(\Gamma, u) \) if the computation succeeds. For fail, \( T(\Gamma, u) \) is defined as usual for all constructions of the \( \lambda \)-calculus; only the \( \cdot \) and \( \ast \) constructions need some additional definitions:

- \((Ax)\) if \( x \) is a variable \( \Gamma \) and \( \Gamma \) contains an assumption \( x : \Phi \) then \( T(\Gamma, x) = (\emptyset, \Phi) \) where \( \emptyset \) is the empty substitution; otherwise \( \Gamma T(\Gamma, x) \) fails;

- \((\Rightarrow E)\) to compute \( T(\Gamma, xu) \) we compute \( (\sigma_1, \Phi_1) = T(\Gamma, x) \) (and we fail if this fails) then we compute \( (\sigma_2, \Phi_2) = T(\Gamma, \sigma_1, u) \) (and fail if this fails) then let \( \alpha \) be a fresh type variable \( \Gamma \) and \( \sigma_2 \) be the most general unifier of \( \Phi_1 \sigma_2 \Rightarrow \alpha \) and \( \Phi_2 \) (we fail if they are not unifiable) then we return \( (\sigma_1 \sigma_2, \gamma) \); and fail it it fails; otherwise we return \( (\sigma, \alpha \sigma \Rightarrow \Phi) \);

- \((\Rightarrow L)\) to compute \( T(\Gamma, x \cdot u) \) we compute \( (\sigma, \Phi) = T((\Gamma, x : \alpha), u) \) where \( \alpha \) is a fresh type variable \( \Gamma \) and fail it it fails; otherwise we return \( (\sigma, \alpha \Rightarrow \Phi) \);
• \((\Box E)\) to compute \(T(\Gamma, u)\Gamma\) we compute \((\sigma_1, \Phi_1) = T(\Gamma, u)\) (and fail if it fails) then we let \(\sigma_2\) be the most general unifier of \(\Phi_1\) and \(\Box \alpha \Gamma\) where \(\alpha\) is a fresh type variable (we fail if this fails) \(\Gamma\) and return \((\sigma_1\sigma_2, \alpha \sigma_2)\).

• \((\Box I)\) to compute \(T(\Gamma, u)\Gamma\) is more complicated. First notice that a sequent \(\Gamma' \vdash u' : \Phi'\) is derivable only when\(\forall\) for all free (term) variables \(x\) in \(u'\) there is an assumption \(x : \Psi\) in \(\Gamma'\) (this is an easy structural induction on the derivation). In particular, if in the case of a term \(\ast \Gamma \sigma \vdash u : \Box \Phi\) can only be derivable if for every free term variable \(x_i \Gamma \leq i \leq n \Gamma \) in \(u\) there is an assumption of the form \(x_i : \Box \Psi_i \) in \(\Gamma\) and \(x_1 : \Box \Psi_1 \sigma, \ldots, x_n : \Box \Psi_n \sigma \vdash u : \Phi\) is derivable.

Therefore, to compute \(T(\Gamma, u)\Gamma\) we first check that all the free variables \(x_1 \Gamma \ldots x_n\) of \(u\) have assumptions \(x_i : \Psi'_i \Gamma \leq i \leq n \Gamma \). We then compute the most general simultaneous unifier \(\sigma_1\) of \(\Psi'_1 \Gamma \ldots \Gamma \Psi'_n\) with \(\Box \alpha_1 \Gamma \ldots \Gamma \Box \alpha_n \Gamma\) where \(\alpha_1 \Gamma \ldots \Gamma \alpha_n\) are fresh type variables \(\Gamma\) and compute \((\sigma_2, \Phi) = T((x_1 : \Psi'_1 \sigma_1, \ldots, x_n : \Psi'_n \sigma_1), u)\). If any operation fails we fail; otherwise, we return \((\sigma_1 \sigma_2, \Box \Phi)\).

The correctness of the algorithm depends on the fact that if \(\Gamma \vdash u : \Phi\) is derivable then \(\Gamma \sigma \vdash u : \Phi \sigma\) is derivable for any substitution \(\sigma\); this follows from an easy induction on the derivation.

Its completeness follows from classical arguments and from the remarks of the \((\Box I)\) case.

That it takes polynomial time follows from the fact that we can compute unifiers of \(\Phi\) and \(\Phi'\) in linear time \(\Gamma\) and represent them as triangular forms of size at most the sum of the sizes of \(\Phi\) and \(\Phi'\) [MM82]. \(\Box\)
References


suppose World-Wide Web servers had Scheme interpreters linked in with their executables. Then small Scheme programs could transfer themselves across the network from server to server to carry out tasks for their masters—perhaps coming back to a home machine to report on their results in the end. Scheme is a good choice for such a language because Scheme implementations are “safe”—it is easy to guarantee that an arbitrary Scheme program received by a server can’t damage the server or compromise its security. With proper cryptographic authentication, an agent can access or commit the user’s resources: buy movie tickets, check his bank account, and so forth.

MIT Artificial Intelligence Lab. HTML page at http://www.ai.mit.edu/projects/su/web-projects.html. Contents: “Suppose World-Wide Web servers had Scheme interpreters linked in with their executables. Then small Scheme programs could transfer themselves across the network from server to server to carry out tasks for their masters—perhaps coming back to a home machine to report on their results in the end. Scheme is a good choice for such a language because Scheme implementations are “safe”—it is easy to guarantee that an arbitrary Scheme program received by a server can’t damage the server or compromise its security. With proper cryptographic authentication, an agent can access or commit the user’s resources: buy movie tickets, check his bank account, and so forth.”

MIT Artificial Intelligence Lab. HTML page at http://www.ai.mit.edu/projects/su/web-projects.html. Contents: “Suppose World-Wide Web servers had Scheme interpreters linked in with their executables. Then small Scheme programs could transfer themselves across the network from server to server to carry out tasks for their masters—perhaps coming back to a home machine to report on their results in the end. Scheme is a good choice for such a language because Scheme implementations are “safe”—it is easy to guarantee that an arbitrary Scheme program received by a server can’t damage the server or compromise its security. With proper cryptographic authentication, an agent can access or commit the user’s resources: buy movie tickets, check his bank account, and so forth.”

MIT Artificial Intelligence Lab. HTML page at http://www.ai.mit.edu/projects/su/web-projects.html. Contents: “Suppose World-Wide Web servers had Scheme interpreters linked in with their executables. Then small Scheme programs could transfer themselves across the network from server to server to carry out tasks for their masters—perhaps coming back to a home machine to report on their results in the end. Scheme is a good choice for such a language because Scheme implementations are “safe”—it is easy to guarantee that an arbitrary Scheme program received by a server can’t damage the server or compromise its security. With proper cryptographic authentication, an agent can access or commit the user’s resources: buy movie tickets, check his bank account, and so forth.”