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@pauillac.inria.fr, Jean.Goubault@ira.uka.de

August 29, 1996

Abstract

A language of constructions for minimal logic is the λ-calculus, where cut-elimination is encoded as β-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 λ-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 λ_{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 λ_{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 λ_{S4}-terms and natural deduction proofs in minimal S4.

However, the λ_{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, formulas-as-types correspondence with the modal logic S4? As already noted by Davies and Pfenning [DP95], the answer is a kind of λ-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 7532.7-06 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.
[†]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 [BdP95], 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 $\lambda\mathtt{ev}Q$-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 $\lambda\sigma$-calculi. This language reveals how evaluation and quoting really work. Interestingly enough, such implementation details such as stacks arise naturally from purely logical principles. The $\lambda\mathtt{ev}Q$-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 reductibility.

In Part III, we prove that the typed $\lambda\mathtt{ev}Q$-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 $\lambda\mathtt{ev}Q$-terms are, as well. The same properties hold with both calculi extended by η-like rules. However, the untyped $\lambda\mathtt{ev}Q$-calculus with η-like rules is not confluent; the status of the untyped $\lambda\mathtt{ev}Q$-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 $\lambda\mathtt{ev}Q$ 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 $\lambda\mu$-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 $\lambda\mathtt{ev}Q$-calculus, an extension of the $\lambda\mathtt{ev}Q$-calculus with η-like rule based on Audebaud's $\lambda\mu$env, 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, the way we derive 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, 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. `()`, or *cons-cells* $(x\ .\ y)$, where x and y are lists or symbols. The list $(x_1\ .\ (x_2\ .\ (\ldots(x_n\ .\ \ nil\ldots)))$ is usually written $(x_1\ \ x_2\ \ \ldots\ \ x_n)$. This universal data structure is not only used to represent ordinary data structures, but also programs: when x_1 evaluates to a function, this list is the program that applies the value of x_1 to the values of $x_2, \ldots, x_n$. Besides, the special lists of the form $(lambda\ \ (x_1\ \ x_2\ \ \ldots\ \ x_n)\ \ t)$ evaluate to the function that takes n arguments, binds them to the symbols $x_1, x_2, \ldots, x_n$ (which act as program variables), 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, `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: $(quote\ \ x)$, also abbreviated as $'x$, leaves x unexecuted, and immediately boxes it as a Lisp data structure, so that executing $'x$ yields x itself, and $(eval\ \ 'x)$ yields the result of the execution of x. (In short, $(eval\ \ 'x) = x$.)

Lisp also has a few other related constructs. The first is `kwote`, which takes an argument x, executes it, and quotes the result. (Therefore, we also have $(eval\ \ (kwote\ \ x)) = x$.) This is useful to build quoted data structures from other quoted data structures. Generalizing `quote` and `kwote`, 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 `kwote` is then operationally the same as `` (lambda (x) `(quote ,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 [CR91], essentially). The correct rule for evaluating a λ-expression, as 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, 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 '(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, 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 as being much simpler than general-purpose programming languages like Lisp or ML, but it isn't necessarily 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, usually) 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 evaluting 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 programming distributed agents.

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, its 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, or 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: $\wedge$ is conjunction, $\vee$ is disjunction, $\Rightarrow$ is implication, $\perp$ is a special constant denoting false; A, B, C, ... are propositional variables, Φ, Ψ, ... are propositional formulas.

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 λ. The *terms* u, v, w, ... in the language are the constants, the applications $f(u_1, \ldots, u_m)$ of m-ary operators f to a list of m terms u_1, ..., u_m, or binding terms like $\lambda x \cdot u$, which bind the variable x in the term u. A *substitution* σ is any map from variables to terms. *Applying* a substitution σ 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 σ and σ' 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 $l \to r$, where l is a term called the left-hand side and t is a term called the right-hand side, with term meta-variables u, v, w, ... that are to be instantiated by arbitrary terms; such a rule is actually a scheme to denote all its *instances* $l\sigma \to r\sigma$. We write $u \longrightarrow v$ or $u \xrightarrow{R} v$ when we wish to make clear which rewrite system we use, to say that u *contracts*, or rewrites in exactly one step, to v; this means that v is obtained from u by replacing a given occurrence of l by r, where $l \to r$ is an instance of a rule in R. We write $u \longrightarrow^* v$ or $u \xrightarrow{R}{}^* v$ to say that u *rewrites* in zero, one or more steps to v, and $u \longrightarrow^+ v$ or $u \xrightarrow{R}{}^+ v$ 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 S4. This is a propositional logic, with additional $\Box$ and $\Diamond$ operators: if Φ is a formula, then $\Box\Phi$ means "necessarily, Φ", or "in all possible futures, Φ"; and $\Diamond\Phi$ means "possibly, Φ", or "there is a future at which Φ". The interpretation in term of possible futures is what makes S4 more specifically a temporal logic. The deduction rules of S4 are usually presented as a Hilbert system:

- Rules:
  - (MP) from Φ and $\Phi \Rightarrow \Psi$, infer Ψ;
  - (Nec) from Φ, 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) $((\Phi \Rightarrow \bot) \Rightarrow \bot) \Rightarrow \Phi$;
  - The modal axioms for S4:
    (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 Φ, Φ_1, Φ_2, Φ_3 range over all formulas. We can define the other connectives as abbreviations: $\neg\Phi = \Phi \Rightarrow \bot$, $\Phi_1 \vee \Phi_2 = \neg\Phi_1 \Rightarrow \Phi_2$, $\Phi_1 \wedge \Phi_2 = \neg(\Phi_1 \Rightarrow \Phi_2 \Rightarrow \bot)$, and $\Diamond\Phi = \neg\Box\neg\Phi$.

This is *classical* S4. We could without doubt provide a functional interpretation for classical S4, but the latter includes classical propositional logic, which has non-obvious functional interpretations by itself. We shall therefore restrict ourselves to intuitionistic versions of S4. To achieve this goal, we don't consider axiom (c) above, and get *minimal* S4. We cannot define the other logical connectives in this system, so 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 $\wedge$ either. This leads us to not only an intuitionistic but in fact a minimal version of S4.

This particular system of intuitionistic S4 with only connective $\Rightarrow$ will simply be called S4 in the sequel. Should we wish to the original logic, we would talk about *classical* S4.

3.2 A Functional Interpretation for S4

As in ordinary Hilbert-style presentations of intuitionistic logic, axioms (s) and (k) are interpreted as the types of the two combinators S and K respectively, and rule (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:

- $Suvw \longrightarrow uw(vw)$;
- $Kuv \longrightarrow u$;

where u, v, w are terms, application is denoted by juxtaposition (i.e., uv means u applied to v) and application is right-associative (so $Suvw$ is really $((Su)v)w$, $uw(vw)$ is really $(uw)(vw)$ and Kuv is really $(Ku)v$).

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

These reduction rules preserve the types, i.e. the theorems proved by a proof represented by $Suvw$ are also proved by the simpler proof $uw(vw)$. 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), (T), (4) and rule (Nec). The basic idea of this work can be explained as follows. If Φ is a type (a proposition), then $\Box\Phi$ is a type of "boxes", containing an object of type Φ. 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 $\Phi_1 \Rightarrow \Phi_2$), and v has type $\Box\Phi_1$ (v is a box containing an argument of type Φ_1), then some term $u \star v$ (where $\star$ has axiom (K) as type, and is written infix for convenience) has type $\Box\Phi_2$. That is, $u \star v$ is a box that contains the application of u to v.

On the other hand, axiom (T) is interpreted as the type of an "unbox" operation ’, and rule (Nec) as the typing rule for the converse operation ‘. We argue that ’ bears strong similarities to Lisp's `eval` primitive, and ‘ looks a lot like Lisp's `quote` special form. Indeed, given a Lisp expression u, (*quote* u) (often abbreviated $'u$) is a (boxed) data structure that, when we apply `eval` on it, yields the value of u. This data structure is built by list building operations, based on the `cons` couple-building primitive, analogous to our $\star$ operation. (But not quite the same, as Lisp does not represent applications uv as couples (u, v), but as $(u, (v, nil))$; this is inessential.) This data structure can also contain boxed representations for λ-expressions, or functions, which we can recreate using combinators: for example, the Lisp expression `'(lambda (x) x)` becomes $‘(SKK)$; we could also build it by hand as $‘S \star ‘K \star ‘K$, and this would be similar to building the Lisp list `(lambda (x) x)` explicitly, by executing the expression `(list 'lambda (list 'x) 'x)`. This suggests the following reduction rules:

- $’(‘u) \longrightarrow u$;
- $’(u \star v) \longrightarrow (’u)(’v)$;
- $‘(uv) \longrightarrow (‘u) \star (‘v)$;

which indeed preserve the types, i.e. are valid proof transformation rules.

There is a dual interpretation of ’ and ‘. For example, $‘((’u)(’v))$ reduces to uv, but Lisp's `'((eval u) (eval v))` does not reduce to anything. Another way of interpreting ‘ and ’ in this case is by means of the backquote notation ` (Scheme's `quasiquote` special form [CR91]) and the comma notation `,` (Scheme's `unquote` special form). We can interpret $‘((’u)(’v))$ as the analogue of ‘(, $u$ , $v$), which is precisely equivalent to ($u$ $v$). Then ‘ would be Lisp's `,` or Scheme's `unquote`, and ’ would be both `eval` and `unquote`. This justifies the following reduction rule, which again preserves the types:

- $‘(’u) \longrightarrow u$;

and which is the equivalent of the Lisp reduction ‘, $u \longrightarrow u$. The fact that we confuse `eval` and `unquote` cannot be considered a departure from the spirit of `eval` and `quote`. However, this kind of rule won't be needed to interpret cut-elimination; at best, we can interpret this as an attempt to an η-like rule in the calculus. (Another would be $‘(uv) \to (‘u) \star (‘v)$.)

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, given any Lisp object, we can quote it on the fly. The operation that does this is called `kwote` in Lisp, and would have type $\Phi \Rightarrow \Box\Phi$. Adding the latter as an axiom would not only entail axiom (4), but it would essentially amount to reduce the logic to pure intuitionistic logic ($\Box\Phi$ being always identical to Φ, we would not need $\Box$ at all). As $\Phi \Rightarrow \Box\Phi$ is in general not provable in S4, we cannot quote any arbitrary object. In particular, the function that maps the variable x to $‘x$ has no meaning; compare the problematic Lisp expression `(lambda (x) (quote x))`, where `x` inside the `quote` expression is understood as *not* being in the scope of the header of the `lambda`-expression.

Axiom (4), on 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 Φ is *boxable* if and only if $\Phi \Rightarrow \Box\Phi$ is provable, i.e. if there is a term $kwote_\Phi$ 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 S4 — i.e., if we consider second-order propositional intuitionistic S4 — so as to be able to speak of data types, then Booleans, integers, products and sums of boxable types are still boxable. But function types are in general not boxable: for example, $A \Rightarrow B$, where A and B are propositional variables is not boxable.

Our interpretation of this phenomenon is the following. In Lisp, `eval` and `quote` mainly have one use that is not subsumed by the use of higher-order functions and closures. This use is *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 $(\mathcal{W}, \mathcal{R}, [\![\_]\!])$, where $\mathcal{W}$ is a non-empty set of so-called *possible worlds* w, $\mathcal{R}$ is a binary relation between worlds called the *accessibility relation* (alternatively, $(\mathcal{W}, \mathcal{R})$ is an oriented graph with $\mathcal{W}$ as set of vertices and $\mathcal{R}$ as set of edges), and $[\![\_]\!]$ 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 $\mathcal{R}$ is a preorder $\geq$. (This preorder is the "future of" relation, in the temporal interpretation.)

The semantics of classical S4 is then given by the relation $w \models \Phi$ (Φ 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 $w' \geq w$, $w' \models \Phi$.

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 proofs, and to translate these proofs as λ-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* , is a set of boxed formulas, that is, of formulas of the form $\Box\Psi$. The restriction that , 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, so as to reflect the functional interpretation of Section 3.2. As is customary, a sequent (a typing judgment) , $\vdash u : \Phi$ consists of a *context* , , which is now a map from variable names x, y, z, ... to formulas (types), and the right hand-side declares that Φ follows from the assumptions , , and that the constructive content of the proof leading to this sequent is u (alternatively, the term u is of type Φ). The language of constructions (of terms) is the λ-calculus [Bar84], augmented with an ’ 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 a ‘ 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, where free occurrence assumes its usual meaning in the λ-calculus. (λ 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 λ-term.

On the other hand, if contexts are maps from variables to types, then , , , $'$ cannot mean set union: we define it as the *overriding* of , by , $'$, i.e. the context that maps x to its type in , $'$ if it exists, and otherwise

$$(Ax) \quad \frac{}{,\ , x : \Phi \vdash x : \Phi}$$

$$(\Rightarrow L) \quad \frac{\begin{array}{c} ,\ , x : \Phi_2 \vdash u : \Phi_3 \\ ,\ \vdash v : \Phi_1 \end{array}}{,\ , y : \Phi_1 \Rightarrow \Phi_2 \vdash u[yv/x] : \Phi_3} \qquad (\Rightarrow R) \quad \frac{,\ , x : \Phi_1 \vdash u : \Phi_2}{,\ \vdash \lambda x \cdot u : \Phi_1 \Rightarrow \Phi_2}$$

$$(\Box L) \quad \frac{,\ , x : \Phi_1 \vdash u : \Phi_2}{,\ , y : \Box\Phi_1 \vdash u[{}^{,}y/x] : \Phi_2} \qquad (\Box R) \quad \begin{array}{c} \dfrac{,\ \vdash u : \Phi}{,\ ,\ ,\ ' \vdash u^{‘} : \Box\Phi} \\ (,\ \text{boxed}) \end{array}$$

$$(Cut) \quad \frac{,\ \vdash u : \Phi_1 \quad ,\ ', x : \Phi_1 \vdash v : \Phi_2}{,\ ,\ ,\ ' \vdash v[u/x] : \Phi_2}$$

Figure 1: System **LS4**

to its type in , 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* $,\ _1 \vdash u_1 : \Phi_1$ *and* $,\ _2 \vdash u_2 : \Phi_2$, *we assume that no variable* x *occurs both in* $,\ _1$ *and in* $,\ _2$.

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 α-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 an multiplicative fashion. This is essential to avoid problems in formulating cut-elimination strategies.

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

$$(Weaken) \quad \frac{,\ \vdash u : \Phi}{,\ ',\ ,\ \vdash u : \Phi}$$

where $,\ '$ is any additional set of typing assumptions $x_i : \Phi_i$, $1 \leq i \leq n$. That is, whenever we have a proof (π) of $,\ \vdash u : \Phi$, there is another proof $,\ ', (\pi)$ of $,\ ',\ ,\ \vdash u : \Phi$. Just add $,\ '$ to the left of all sequents in (π) 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, at least 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, that is to apply cut-elimination, not just to wish cuts were not there, 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, as 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: Φ in (Ax), $\Phi_1 \Rightarrow \Phi_2$ in $(\Rightarrow L)$ and $(\Rightarrow R)$, $\Box\Phi$ in $(\Box L)$ and $(\Box R)$.

Given two sub-proofs (π_1) and (π_2), performing a cut on their end sequents yields a proof of the form:

$$\begin{array}{rc} & (\pi_1) \qquad\qquad\qquad (\pi_2) \\ & \vdots \qquad\qquad\qquad\quad \vdots \\ (Cut) & \dfrac{,_1 \vdash u_1 : \Psi_1 \quad ,_2, x_1 : \Psi_1 \vdash u_2 : \Psi_2}{,_1, ,_2 \vdash u_2[u_1/x_1] : \Psi_2} \end{array}$$

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

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

We now proceed to permuting cuts up, and there are two cases.

4.1.1 Case 1: Ψ_1 is active on both sides

In the first case, Ψ_1 is active both on the right-hand side of $,_1 \vdash u_1 : \Psi_1$ and on the left-hand side of $,_2, x_1 : \Psi_1 \vdash u_2 : \Psi_2$. We have the following cases, of which the first one is degenerate:

(Ax) $,_1 = ,, x : \Phi$, $u_1 = x$ and $\Psi_1 = \Phi$; we tranform:

$$\begin{array}{rc} & \qquad\qquad\qquad\qquad (\pi_2) \\ & \qquad\qquad\qquad\qquad \vdots \\ \begin{array}{r}(Ax)\\(Cut)\end{array} & \dfrac{\dfrac{}{,, x : \Phi \vdash x : \Phi} \quad ,_2, x_1 : \Phi \vdash u_2 : \Psi_2}{,, x : \Phi, ,_2 \vdash u_2[x/x_1] : \Psi_2} \end{array}$$

into:

$$\begin{array}{c} ,, (\pi_2)[x/x_1] \\ \vdots \\ ,, x : \Phi, ,_2 \vdash u_2[x/x_1] : \Psi_2 \end{array}$$

where $,, (\pi_2)[x/x_1]$ is the result of replacing x_1 by x throughout (π_2) — this effectively *contracts* two assumptions for Φ into one — and then weakening $(\pi_2)[x/x_1]$ by $,$.

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

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

$$\begin{array}{rc} & (\pi_1') \qquad\qquad\qquad\qquad (\pi_2') \qquad\qquad (\pi_2'') \\ & \vdots \qquad\qquad\qquad\qquad\quad \vdots \qquad\qquad\quad \vdots \\ \begin{array}{r}(\Rightarrow R)\\(Cut)\end{array} & \dfrac{\dfrac{,_1, x : \Phi_1 \vdash u : \Phi_2}{,_1 \vdash \lambda x \cdot u : \Phi_1 \Rightarrow \Phi_2} \quad (\Rightarrow L)\ \dfrac{,_2, z : \Phi_2 \vdash v : \Phi_3 \quad ,_2 \vdash w : \Phi_1}{,_2, y : \Phi_1 \Rightarrow \Phi_2 \vdash v[yw/z] : \Phi_3}}{,_1, ,_2 \vdash v[yw/z][\lambda x \cdot u/y] : \Phi_3} \end{array}$$

into:

$$\dfrac{(Cut)\;\dfrac{\begin{matrix}(\pi_1')\\ \vdots\\ ,_1, x : \Phi_1 \vdash u : \Phi_2\end{matrix} \quad \begin{matrix}(\pi_2')\\ \vdots\\ ,_2, z : \Phi_2 \vdash v : \Phi_3\end{matrix}}{,_1, x : \Phi_1, ,_2 \vdash v[u/z] : \Phi_3} \quad \begin{matrix}(\pi_2'')\\ \vdots\\ ,_2 \vdash w : \Phi_1\end{matrix}}{,_1, ,_2 \vdash v[u/z][w/x] : \Phi_3}\;(Cut)$$

Because of the variable naming convention, $v[yw/z][\lambda x \cdot u/y]$ is in fact $v[(\lambda x \cdot u)w/z]$, and $v[u/z][w/x]$ is $v[u[w/x]/z]$, so cut-elimination here means transforming $v[(\lambda x \cdot u)w/z]$ into $v[u[w/x]/z]$, and this is precisely the β-reduction rule.

$(\Box R)/(\Box L)$ $,_1 = ,, ,'$ where $,$ is boxed, $u_1 = v‘$, $\Psi_1 = \Box\Phi$, $x_1 = y$, $u_2 = u[’y/x]$, $\Psi_2 = \Phi_2$, and we transform:

$$(Cut)\;\dfrac{\dfrac{\begin{matrix}(\pi_1')\\ \vdots\\ , \vdash v : \Phi\end{matrix}}{,, ,' \vdash v‘ : \Box\Phi} \quad \dfrac{\begin{matrix}(\pi_2')\\ \vdots\\ ,_2, x : \Phi \vdash u : \Phi_2\end{matrix}}{,_2, y : \Box\Phi \vdash u[’y/x] : \Phi_2}}{,, ,', ,_2 \vdash u[’y/x][v‘/y] : \Phi_2}$$

into:

$$(Cut)\;\dfrac{\begin{matrix},', (\pi_1')\\ \vdots\\ ,, ,' \vdash v : \Phi\end{matrix} \quad \begin{matrix}(\pi_2')\\ \vdots\\ ,_2, x : \Phi \vdash u : \Phi_2\end{matrix}}{,, ,', ,_2 \vdash u[v/x] : \Phi_2}$$

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

4.1.2 Case 2: Ψ_1 is inactive on some side

In the second case, Ψ_1 is not active on the right-hand side of $,_1 \vdash u_1 : \Psi_1$ (then (π_1) ends with a rule acting on the left or with (Cut)) or Ψ_1 is not active on the left-hand side of $,_2, x_1 : \Psi_1 \vdash u_2 : \Psi_2$. What happens here is that we can always permute (Cut) with the rule above it where Ψ_1 is not active, at least when this rule is not $(\Box R)$ with Φ_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 $,_1 = ,, y : \Phi_1 \Rightarrow \Phi_2$, $u_1 = u[yv/x]$, $\Psi_1 = \Phi_3$, and we transform:

$$\dfrac{(\Rightarrow L)\;\dfrac{\begin{matrix}(\pi_1')\\ \vdots\\ ,, x : \Phi_2 \vdash u : \Phi_3\end{matrix} \quad \begin{matrix}(\pi_1'')\\ \vdots\\ , \vdash v : \Phi_1\end{matrix}}{,, y : \Phi_1 \Rightarrow \Phi_2 \vdash u[yv/x] : \Phi_3} \quad \begin{matrix}(\pi_2)\\ \vdots\\ ,_2, x_1 : \Phi_3 \vdash u_2 : \Psi_2\end{matrix}}{,, y : \Phi_1 \Rightarrow \Phi_2, ,_2 \vdash u_2[u[yv/x]/x_1] : \Psi_2}\;(Cut)$$

into:

$$\dfrac{(Cut)\;\dfrac{\begin{matrix}(\pi_1')\\ \vdots\\ ,, x : \Phi_2 \vdash u : \Phi_3\end{matrix} \quad \begin{matrix}(\pi_2)\\ \vdots\\ ,_2, x_1 : \Phi_3 \vdash u_2 : \Psi_2\end{matrix}}{,, x : \Phi_2, ,_2 \vdash u_2[u/x_1] : \Psi_2} \quad \begin{matrix},_2, (\pi_1'')\\ \vdots\\ ,, ,_2 \vdash v : \Phi_1\end{matrix}}{,, y : \Phi_1 \Rightarrow \Phi_2, ,_2 \vdash u_2[u/x_1][yv/x] : \Psi_2}\;(\Rightarrow L)$$

Computationally speaking, this rewrites $u_2[u[yv/x]/x_1]$ into $u_2[u/x_1][yv/x]$, 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 [ACCL90].)

The only problem occurs when the last rule of (π_2) is $(\Box R)$—in particular, Ψ_1 is inactive on the left-hand side of $, _2, x_1 : \Psi_1 \vdash u_2 : \Psi_2$. The cut then looks like:

$$(Cut)\ \dfrac{\begin{array}{c}(\pi_1)\\ \vdots\\ , _1 \vdash u_1 : \Psi_1\end{array} \quad (\Box R)\ \dfrac{\begin{array}{c}(\pi_2')\\ \vdots\\ , \vdash u : \Phi\end{array}}{, , , ', x : \Psi_1 \vdash u^{‘} : \Box\Phi}}{, _1, , , , ' \vdash u^{‘}[u_1/x] : \Box\Phi}$$

where , is boxed, and (unless $, _1$ is boxed) we cannot transform this into the following:

$$\begin{array}{l} (Cut) \\ (\Box R) \end{array}\ \dfrac{\dfrac{\begin{array}{c}(\pi_1)\\ \vdots\\ , _1 \vdash u_1 : \Psi_1\end{array} \quad \begin{array}{c}x : \Psi_1, (\pi_2')\\ \vdots\\ , , x : \Psi_1 \vdash u : \Phi\end{array}}{, , , _1 \vdash u[u_1/x] : \Phi}}{, , , _1, , ' \vdash (u[u_1/x])^{‘} : \Box\Phi}$$

which is indeed not a proof at all if $, _1$ is not boxed.

So, we have to find another way, and examine the last rule we used in (π_1). There are several cases and subcases:

- Either Ψ_1 is inactive on the right of the rule, and we just bubble the (Cut) up inside (π_1). This has no effect on the resulting construction: for example, if the last rule of (π_1) is $(\Rightarrow L)$, then u_1 has the form $v_1[yv_2/z]$, so the construction before the cut is pushed upwards is $u^{‘}[v_1[yv_2/z]/x]$, and the construction afterwards is $u^{‘}[v_1/x][yv_2/z]$, which is the same, because of the variable naming convention. Note that if the last rule of (π_1) is another (Cut), pushing the lower cut upwards does not change the construction either.

- Or Ψ_1 is active on the right of the rule, and we again have two cases, according to whether Ψ_1 is a boxed formula or not:

  – **Case (a)** if Ψ_1 is not a boxed formula, then x cannot occur free in u, since the assumption $x : \Psi_1$ cannot be used to type u in (π_2'). We can therefore transform the proof into:

  $$(\Box R)\ \dfrac{\begin{array}{c}(\pi_2')\\ \vdots\\ , \vdash u : \Phi\end{array}}{, _1, , , , ' \vdash u^{‘} : \Box\Phi}$$

  This suggests the identity $u^{‘}[u_1/x] = u^{‘}$ 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 Ψ_1 is a boxed formula $\Box\Phi_1$, we can assume that the end sequent of (π_2') contains an assumption of the form $x : \Box\Phi_1$ on its left-hand side, and we can therefore bubble up the (Cut) above the $(\Box R)$ rule on the right. Notice that the only ways that $\Box\Phi_1$ can be active on the right-hand side of the end sequent of (π_1) are that the last rule of (π_1) be either (Ax) or $(\Box R)$. This yields two sub-cases:

  Case (b) The last rule of (π_1) is (Ax), and the proof looks like:

  $$\begin{array}{l} (Ax) \\ (Cut) \end{array}\ \dfrac{\dfrac{}{, _1, y : \Box\Phi_1 \vdash y : \Box\Phi_1} \quad (\Box R)\ \dfrac{\begin{array}{c}(\pi_2')\\ \vdots\\ , , x : \Box\Phi_1 \vdash u : \Phi\end{array}}{, , , ', x : \Box\Phi_1 \vdash u^{‘} : \Box\Phi}}{, _1, , , , ', y : \Box\Phi_1 \vdash u^{‘}[y/x] : \Box\Phi}$$

$$(Ax)\quad \dfrac{}{,\ , x : \Phi \vdash x : \Phi}$$

$$(\Rightarrow I)\quad \dfrac{,\ , x : \Phi_1 \vdash u : \Phi_2}{,\ \vdash \lambda x \cdot u : \Phi_1 \Rightarrow \Phi_2} \qquad (\Rightarrow E)\quad \dfrac{\begin{array}{c}, \vdash u : \Phi_1 \Rightarrow \Phi_2 \\ , \vdash v : \Phi_1\end{array}}{, \vdash uv : \Phi_2}$$

$$(\Box I)\quad \begin{array}{c}\dfrac{, \vdash u : \Phi}{,\ ,\ ,' \vdash u^{‘} : \Box\Phi} \\ (,\ \text{boxed})\end{array} \qquad (\Box E)\quad \dfrac{, \vdash u : \Box\Phi}{, \vdash {}^{›}u : \Phi}$$

Figure 2: Natural Deduction for S4

and we transform it into:

$$\begin{array}{rc} & (\pi_2')[y/x] \\ & \vdots \\ (\Box R) & \dfrac{,\ , y : \Box\Phi_1 \vdash u[y/x] : \Phi}{,\ _1, ,\ ,\ ,', y : \Box\Phi_1 \vdash (u[y/x])^{‘} : \Box\Phi}\end{array}$$

This suggests that the free variables of $u^{‘}$ should in fact be the same as those of u.
Case (c) The last rule of (π_1) is $(\Box R)$, and the proof looks like:

$$\begin{array}{rc} & \begin{array}{cc} (\pi_1') & (\pi_2') \\ \vdots & \vdots \end{array} \\ \begin{array}{r}(\Box R) \\ (Cut)\end{array} & \dfrac{\dfrac{,\ _1' \vdash v_1 : \Phi_1}{,\ _1 \vdash v_1^{‘} : \Box\Phi_1} \quad (\Box R)\ \dfrac{,\ , x : \Box\Phi_1 \vdash u : \Phi}{,\ ,\ ,', x : \Box\Phi_1 \vdash u^{‘} : \Box\Phi}}{,\ _1, ,\ ,\ ,' \vdash u^{‘}[v_1^{‘}/x] : \Box\Phi}\end{array}$$

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

$$\begin{array}{rc} & \begin{array}{cc} (\pi_1') & \\ \vdots & (\pi_2') \\ & \vdots \end{array} \\ \begin{array}{r}(\Box R) \\ (Cut) \\ (\Box R)\end{array} & \dfrac{\dfrac{\dfrac{,\ _1' \vdash v_1 : \Phi_1}{,\ _1' \vdash v_1^{‘} : \Box\Phi_1} \quad ,\ , x : \Box\Phi_1 \vdash u : \Phi}{,\ _1', , \vdash u[v_1^{‘}/x] : \Phi}}{,\ _1, ,\ ,\ ,' \vdash (u[v_1^{‘}/x])^{‘} : \Box\Phi}\end{array}$$

Should we choose $u^{‘} = {}^{‘}u$ as a definition, this would be automatically satisfied.

In short, 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, and ignore the proviso that some cuts cannot be pushed up $(\Box R)$ right away:

Theorem 4.1 *Let the λ›‘-calculus be the λ-calculus with operations ›, ‘, with the following reduction rules:*

- $(\lambda x \cdot u)v \to u[v/x]$
- $,(‘u) \to u$

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* **LS4** *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 $u \to v$, 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 $z : \Box A$, $x : B \Rightarrow \Box A$, $y : B$. Its most general type is $(B \Rightarrow \Box A) \Rightarrow B \Rightarrow \Box\Box A$, but it reduces to $\lambda x \cdot \lambda y \cdot ‘(xy)$, which is not typable under the assumptions $x : B \Rightarrow \Box A$ and $y : B$. In fact, this term, 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 $‘$), but then xy cannot be typed. $\Box$

The termination proof extends to any type system for the pure λ-calculus such that typable terms are strongly normalizing. This includes Girard's System F [Gir71, GLT89], Krivine and Leivant's AF2 [Kri92], 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 λ-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 λ-calculus.

Notice that the typing rules of Figure 2 are nothing but a reformulation of **LS4** 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 (Cut). Hence, any construction in **LS4** is a typable $\lambda,‘$-term.

Because of the failure of subject reduction, we cannot conclude directly that the cut-elimination process terminates; indeed, reduction may terminate on an untypable 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 **LS4** 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 morale of Section 4.1 is the following: $u^‘$ should behave as $‘u$, in as much as $,u^‘$ must rewrite to u, but it cannot be built from u by just applying some operators to u. In fact, it appears that $‘$ 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 — , 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 $‘u$, but something like $‘_{[x_1:=x_1,\ldots,x_n:=x_n]}u$, 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 $‘_{[x_1:=x_1,\ldots,x_n:=x_n]}u$, therefore blocking substitutions from affecting u itself. Indeed, $u^‘[u_1/x_i]$ now means $‘_{[x_1:=x_1,\ldots,x_{i-1}:=x_{i-1},x_i:=u_1,x_{i+1}:=x_{i+1},\ldots,x_n:=x_n]}u$, which is different from $(u[u_1/x])^‘$.

This is the solution of Bierman and de Paiva [BdP95] (except that they also consider sums, i.e. disjunctions). Their notation for $‘_{[x_1:=v_1,\ldots,x_n:=v_n]}u$ is $\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n$, where the variables after the for keyword are considered bound, and $\text{‚}u$ is written unbox u, with the following sole rewrite rule on 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, $u^{\text{‘}}$ is in fact encoded as follows (we make this a definition, since we shall use it in the rest of this section):

Definition 4.1 *For any term u, with free variables x_1, ..., x_n, we define $u^{\text{‘}}$ 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, ..., y_n are n pairwise distinct fresh variables.

The variables y_i, $1 \leq i \leq n$, are the bound variables of the construct, and are subject to α-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 $u^{\text{‘}}[v_1/x_1, \ldots, v_n/x_n]$.

However, this calculus is not satisfactory. Indeed, the latter rule deals with the case of a cut-elimination in the $(\Box R)/(\Box L)$ 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 $(\Box R)$ rule, as Bierman and de Paiva do, for the following compound rule $(\Box \mathcal{I})$:

$$\frac{, \vdash v_1 : \Box\Phi_1 \quad \ldots \quad , \vdash v_n : \Box\Phi_n \quad x_1 : \Box\Phi_1, \ldots, x_n : \Box\Phi_n \vdash u : \Phi}{, \vdash \text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n : \Box\Phi}$$

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, say, automated deduction, where the subformula property is crucial.

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

$$(Cut)\ \dfrac{, \vdash v_1 : \Box\Phi_1 \qquad \begin{array}{c} (Cut)\ \dfrac{, \vdash v_n : \Box\Phi_n \quad (\Box R)\ \dfrac{x_1 : \Box\Phi_1, \ldots, x_n : \Box\Phi_n \vdash u : \Phi}{x_1 : \Box\Phi_1, \ldots, x_n : \Box\Phi_n \vdash u^{\text{‘}} : \Box\Phi}}{x_1 : \Box\Phi_1, \ldots, x_{n-1} : \Box\Phi_{n-1} u^{\text{‘}}[v_n/x_n] : \Box\Phi} \\ \vdots \end{array}}{, \vdash u^{\text{‘}}[v_1/x_1, \ldots, v_n/x_n] : \Box\Phi}$$

and then Bierman and de Paiva's rule is not enough to eliminate all cuts, since for example $(\Box R)/(\Box R)$ cuts (case (c) in Section 4.1.2) are not eliminated: the prototypical example is box x with (box u with ϵ for ϵ) for x (where we write ϵ 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 $(\Box R)/(\Box R)$ cut.

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

$$\begin{array}{l} \text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n \\ \rightarrow \text{box } u[v_i'^{\text{‘}}/x_i] \text{ with } v_1, \ldots, v_{i-1}, w_1, \ldots, w_m, v_{i+1}, \ldots, v_n \text{ for } x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n \end{array}$$

when v_i has the form box v_i' with $w_1, \ldots, w_m$ for $y_1, \ldots, y_m$ (we say that v_i is a box-term).

But cases (a) and (b) also pose problems of their own. Case (a) demands that all the free variables of $u^{\text{‘}}$ are free in u, or at least that $u^{\text{‘}}$ rewrites to some term having no more free variables than u. This suggests adding a *garbage collection rule*:

$$\begin{array}{l} \text{box } u \text{ with } v_1, \ldots, v_n \text{ for } x_1, \ldots, x_n \\ \rightarrow \text{box } u \text{ with } v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \text{ for } x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \end{array}$$

$$\begin{array}{ll}
(\beta) & (\lambda x \cdot u)v \to u[v/x] \\
(\mathsf{unbox}\ ) & \mathsf{unbox}\ (\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n) \to u[v_1/x_1, \ldots, v_n/x_n] \\
(\mathsf{box}) & \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n \\
 & \quad \to \mathsf{box}\ u[v_i'^{\text{‘}}/x_i] \quad \mathsf{with}\ v_1, \ldots, v_{i-1}, w_1, \ldots, w_m, v_{i+1}, \ldots, v_n \\
 & \qquad\qquad\qquad\quad\ \mathsf{for}\ x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n \\
 & \text{if } v_i = \mathsf{box}\ v_i'\ \mathsf{with}\ w_1, \ldots, w_m\ \mathsf{for}\ y_1, \ldots, y_m \\
(\mathsf{gc}) & \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n \\
 & \to \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \\
 & \text{if } x_i \text{ is not free in } u \\
(\mathsf{ctract}) & \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n \\
 & \to \mathsf{box}\ u[x_i/x_j]\ \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n \\
 & \text{if } i \neq j, v_i = v_j
\end{array}$$

Commuting conversion:

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

for any permutation π of $\{1, \ldots, n\}$

Figure 3: Reduction in the λ_{S4}-Calculus

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 $u^{\text{‘}}$. This places the restriction on terms of the form $\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n$ that any free variable of u is some x_i, $1 \leq i \leq n$. 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:

$$\begin{array}{l}
\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n \\
\to \mathsf{box}\ u[x_i/x_j]\ \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n
\end{array}$$

whenever $i \neq j$ and $v_i = v_j$.

Another symmetry that is present in substitutions gives rise to so-called *commuting conversions*, which represent on the level of terms the fact that we can always permute cuts with each other on the level of proofs. In natural deduction, this means that we can permute all minor premises (of $(\Box\mathcal{I})$, here) without changing the proof. That is, for any permutation π 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:

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

and

$$\mathsf{box}\ u\ \mathsf{with}\ v_{\pi(1)}, \ldots, v_{\pi(n)}\ \mathsf{for}\ x_{\pi(1)}, \ldots, x_{\pi(n)}$$

as being equivalent. This is consistent with the previous rewrite rules.

To recap, we define:

Definition 4.2 (λ_{S4}) *The λ_{S4}-calculus is defined by the following syntax, where x, y, z, ... denote variables taken from an infinite set $\mathcal{V}$. Terms s, t, u, v, w, ... are elements of the language T defined by:*

$$\begin{array}{rcl}
T & ::= & \mathcal{V} \mid \lambda\mathcal{V} \cdot T \mid TT \\
 & \mid & \mathsf{unbox}\ T \mid \mathsf{box}\ T\ \mathsf{with}\ T, \ldots, T\ \mathsf{for}\ \mathcal{V}, \ldots, \mathcal{V}
\end{array}$$

$$(Ax)\quad \frac{}{\ ,\ , x : \Phi \vdash x : \Phi}$$

$$(\Rightarrow I)\quad \frac{\ ,\ , x : \Phi_1 \vdash u : \Phi_2}{\ ,\ \vdash \lambda x \cdot u : \Phi_1 \Rightarrow \Phi_2} \qquad (\Rightarrow E)\quad \frac{\begin{array}{c} ,\ \vdash u : \Phi_1 \Rightarrow \Phi_2 \\ ,\ \vdash v : \Phi_1 \end{array}}{\ ,\ \vdash uv : \Phi_2}$$

$$(\Box\mathcal{I})\quad \frac{\begin{array}{c} ,\ \vdash v_1 : \Box\Phi_1 \quad \ldots \quad ,\ \vdash v_n : \Box\Phi_n \\ x_1 : \Box\Phi_1, \ldots, x_n : \Box\Phi_n \vdash u : \Phi \end{array}}{\ ,\ \vdash \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n : \Box\Phi} \qquad (\Box\mathcal{E})\quad \frac{\ ,\ \vdash u : \Box\Phi}{\ ,\ \vdash \mathsf{unbox}\ u : \Phi}$$

Figure 4: Typing Rules for the λ_{S4}-Calculus

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

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

is such that $\mathrm{fv}(u) \subseteq \{x_1, \ldots, x_n\}$*, and* $\mathrm{fv}(v_i) \cap \{x_1, \ldots, x_n\} = \emptyset$ *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* α*-convertible, then* $u \sim v$*. Unless stated otherwise, we always consider* λ_{S4}*-terms modulo* $\sim$*.*

The typing rules are given in Figure 4.

We shall sometimes write box u with σ instead of the more cumbersome 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 (α-conversion and commuting conversion), so the calculus is well-defined.

5 Properties of the λ_{S4}-Calculus

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

Theorem 5.1 *The properties of the* λ_{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* **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 λ_{S4}-calculus is, in fact, not quite satisfactory.

5.1 Proofs of the Properties

The most natural way to prove that the λ_{S4}-calculus is confluent is to first prove the finiteness of developments, as in [Bar84], 11.2, for the case of the pure λ-calculus.

Definition 5.1 ($\lambda_{S4}{}'$) *We define the* $\lambda_{S4}{}'$-calculus *as follows.*

The terms u, v, w, ... are variables x, y, z, ..., applications uv, abstractions $\lambda x \cdot u$ and primed redexes $(\lambda' x \cdot u)v$ (we then consider all free occurrences of x in u as bound), evaluations unbox u *and quotations* 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 λ_{S4}-terms.*

Reduction in the $\lambda_{S4}{}'$-calculus is defined as in λ_{S4}, except that (β) is replaced by:

$$(\beta')\ (\lambda' x \cdot u)v \to u[v/x]$$

The erasing transformation E from $\lambda_{S4}{}'$ to λ_{S4} consists in erasing all prime symbols from λ'-redexes (i.e., $E((\lambda' x \cdot u)v) = (\lambda x \cdot E(u))E(v)$, in particular.)

Notice that primed redexes are *not* considered as applications, but rather as entirely new expressions; think of $(\lambda' x \cdot u)v$ as another notation for, say, let $x = v$ in u.

Our aim is to show that the $\lambda_{S4}{}'$-calculus terminates. To do so, we define an appropriate notion of weighting of variables and terms:

Definition 5.2 *We define the $\lambda_{S4}{}'^{*}$-calculus as the following extension of the $\lambda_{S4}{}'$-calculus.*

The terms are as in the $\lambda_{S4}{}'$-calculus, except that each variable can be either a weighted variable x^k, *where k is a positive integer, or a* proxy variable $\hat{x}$, $\hat{y}$, ...

More precisely, the $\lambda_{S4}{}'^{*}$-terms u, v, w, ... are proxy variables $\hat{x}$, $\hat{y}$, ..., weighted variables x^k, y^k, z^k, ... $(k > 0)$, applications uv, abstractions $\lambda x \cdot u$ and primed redexes $(\lambda' x \cdot u)v$ (we then consider all free occurrences of x^k, for any $k > 0$, in u as bound), evaluations unbox u *and quotations* 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 $\hat{x}$ in u are considered bound in the* box*-term.) These terms are subject to the same provisos and variable naming convention than λ_{S4}-terms. Moreover, we impose that no variable x occurs both as a proxy $\hat{x}$ and as a weighted variable x^k.*

Well-formed $\lambda_{S4}{}'^{*}$-terms *are those $\lambda_{S4}{}'^{*}$-terms such that every occurrence of a proxy variable is bound by some* box*-expression.*

Substitution in the $\lambda_{S4}{}'^{*}$-calculus is defined so that weights and proxy status are ignored (i.e., in particular $x^k[u/x] = \hat{x}[u/x] = u$, $x^k[u/y] = x^k$ and $\hat{x}[u/y] = \hat{x}$ if $x \neq y$).

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

Reduction in the $\lambda_{S4}{}'^{*}$-calculus is then defined as in λ_{S4}', except for the change of meaning of substitution, and for the following reformulation of the rules (box) *and* (ctract)*:*

$$\begin{array}{ll} (\mathsf{box}') & \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_n \\ & \to \mathsf{box}\ u[\mathsf{box}\ v\ \mathsf{with}\ \hat{z}_1, \ldots, \hat{z}_m\ \mathsf{for}\ y_1, \ldots, y_m/x_i] \\ & \quad \mathsf{with}\ v_1, \ldots, v_{i-1}, w_1, \ldots, w_m, v_{i+1}, \ldots, v_n \\ & \quad \mathsf{for}\ x_1, \ldots, x_{i-1}, z_1, \ldots, z_m, x_{i+1}, \ldots, x_n \end{array}$$

if $v_i =$ box v with $w_1, \ldots, w_m$ for $y_1, \ldots, y_m$, *and where z_1, ..., z_m are m fresh pairwise distinct proxy variables,*

$$\begin{array}{ll} (\mathsf{ctract}') & \mathsf{box}\ u\ \mathsf{with}\ \ldots, v_i, \ldots, v_j, \ldots\ \mathsf{for}\ \ldots, x_i, \ldots, x_j, \ldots \\ & \to \mathsf{box}\ u\{x_i/x_j\}\ \mathsf{with}\ \ldots, v_i, \ldots, v_{j-1}, v_{j+1}, \ldots\ \mathsf{for}\ \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots \end{array}$$

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

The erasing transformation E_0 from $\lambda_{S4}{}'^{*}$ to λ_{S4}' consists in erasing all weights and proxy statuses (i.e., in particular $E_0(x^k) = E_0(\hat{x}) = x$).

Given a term u, if x only occurs as a proxy $\hat{x}$ in u, then we shall say that x itself is a proxy variable; otherwise, by our conventions x can only occur weighted in u, 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 α-renaming (and E_0-erasing). The idea in using proxy variables is that in the formulation for rule (box') above, the proxies $\hat{z}_j$, $1 \leq j \leq m$, are not meant to have weights of their own, but rather to represent w_j, to which they are bound by the outer box term.

We define the *weight* of a well-formed $\lambda_{\mathrm{S4}}{'}^{*}$-term as the sum of the weights of variables occurring (free or bound) in it, where the weight of x^k is k, and the weight of $\hat{x}$ is recursively defined as the weight of the term for which it is a proxy. Formally:

Definition 5.3 *For every $\lambda_{\mathrm{S4}}{'}^{*}$-term u, for every function ω mapping proxy variables to positive integers, define the* weight $W(u, \omega)$ *by structural induction as follows:*

- $W(x^k, \omega) = k$;
- $W(\hat{x}, \omega) = \omega(x)$;
- $W(uv, \omega) = W((\lambda' x \cdot u)v, \omega) = W(u, \omega) + W(v, \omega)$;
- $W(\lambda x \cdot u, \omega) = W(\mathsf{unbox}\ u, \omega) = W(u, \omega)$;
- $W(\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n) = W(u, \omega') + \sum_{i=1}^{n} W(v_i, \omega)$, *where ω' 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 $\lambda_{\mathrm{S4}}{'}^{*}$-term u is independent of the particular function ω 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 $\lambda_{\mathrm{S4}}{'}^{*}$-term u relatively to u, which we shall write $W_u(v)$.

Notice also that, because addition is associative and commutative, Definition 5.3 makes sense modulo the commutative conversion $\sim$.

As in [Bar84], we can see $\lambda_{\mathrm{S4}}{'}^{*}$-terms as $\lambda_{\mathrm{S4}}{'}$-terms, plus an additional mapping from occurrences of variables to either a positive integer k (to represent x^k) or to a special proxy token (to represent $\hat{x}$). Such a mapping is called a *weighting*; terms that have the same E_0-erasure only differ by their weightings.

Definition 5.4 *We say that the weighting of a well-formed $\lambda_{\mathrm{S4}}{'}^{*}$-term t is* decreasing *if and only if:*

- *for every subterm of t of the form $(\lambda' x \cdot u)v$, for every free occurrence x^k of x in u, $k > W_t(v)$;*
- *and for every subterm of t of the form:*

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

  for every free occurrence x_i^k of the non-proxy variable x_i in u, where $1 \leq i \leq n$, $k > W_t(v_i)$.

Lemma 5.2 *For every $\lambda_{\mathrm{S4}}{'}$-term u_0, u_0 admits a decreasing weighting; that is, there is a well-formed $\lambda_{\mathrm{S4}}{'}^{*}$-term u such that $E_0(u) = u_0$, and such that the weighting of u is decreasing.*

Proof: Number the occurrences of variables in u_0 from right to left, giving the nth 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 $W_u(v)$ of any subterm v of u is just the sum of the weights k of variable occurrences x^k in v, and the result is clear. (This is as in [Bar84].) $\square$

Lemma 5.3 *Let u and v be two $\lambda_{\mathrm{S4}}{'}^{*}$-terms, such that u is well-formed, the weighting of u is decreasing, and u 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

$$\mathsf{unbox}\ (\mathsf{box}\ s\ \mathsf{with}\ t_1, \ldots, t_n\ \mathsf{for}\ z_1, \ldots, z_n)$$

where every z_i, $1 \leq i \leq n$, is a proxy variable.

(iii) The weighting of v is decreasing.

Proof: Let Δ_1 be the contracted redex in u.

(i) is clear; observe that all the fresh proxy variables $z_1, \ldots, z_m$ in rule (box') are indeed bound by the outer box-term.

(ii) If Δ_1 is a (β') redex $(\lambda' x \cdot s)t$, 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, the λ'-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 Δ_1 is an (unbox) redex:

$$\mathsf{unbox}\ (\mathsf{box}\ s\ \mathsf{with}\ t_1, \ldots, t_n\ \mathsf{for}\ x_1, \ldots, x_n)$$

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

If Δ_1 is a (box') redex:

$$\mathsf{box}\ s\ \mathsf{with}\ t_1, \ldots, t_i, \ldots, t_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_n$$

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

- Either x_i is a weighted variable, and:
  * on the one hand, zero, one or more occurrences of x_i in s are replaced by

    $$\mathsf{box}\ t\ \mathsf{with}\ \hat{z}_1, \ldots, \hat{z}_m\ \mathsf{for}\ y_1, \ldots, y_m$$

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

    $$s[\mathsf{box}\ t\ \mathsf{with}\ \hat{z}_1, \ldots, \hat{z}_m\ \mathsf{for}\ y_1, \ldots, y_m / x_i]$$

    in v is at most that of s in u.
  * On the other hand, t_i is replaced by $w_1, \ldots, w_m$ in the with part of Δ_1, but by definition of t_i, the weight of t_i is strictly greater than the sum of the weights of $w_1, \ldots, w_m$.
    So $W(u) > W(v)$.
- Or x_i is a proxy, and on the one hand the weight of $s[\mathsf{box}\ t\ \mathsf{with}\ \hat{z}_1, \ldots, \hat{z}_m\ \mathsf{for}\ y_1, \ldots, y_m / x_i]$ in v is exactly the same as that of s in u; and on the other hand, t_i is replaced by $w_1, \ldots, w_m$ in the with part of Δ_1, so that again the weight decreases strictly.

If Δ_1 is a (gc) redex, then the weight decreases strictly, because one term of positive weight vanishes.

If Δ_1 is a (ctract') redex $\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n$, with $v_i = v_j$, then contracting it means replacing u by $u\{x_i/x_j\}$, whose weight does not change, and deleting v_j, which makes the overall weight decrease.

(iii) Let Δ_0 be a λ'-redex or a box term occurring in v. Δ_0 is the residual of some λ'-redex (resp. of some box term) Δ_2 in u. We prove the claim by case analysis on the respective positions of Δ_1 and Δ_2 in u. The only non-trivial cases are:

- If Δ_2 is $(\lambda' x \cdot u')v'$ (resp. $\mathsf{box}\ u'\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n$, 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 Δ_1 lies inside v', then by (i) Δ_1 is contracted to a term with weight at most that of Δ_1; therefore v' reduces during this contraction to a term with at most the same weight, and every occurrence x^k of x of u' remains of greater weight than that of the contracted v'.

– If Δ_1 is $(\lambda' x \cdot u')v'$ (resp. $\mathsf{box}\ u'\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n$, with $x = x_i$ a non-proxy variable for some arbitrary i, $1 \leq i \leq n$, in which case we let $v' = v_i$) and Δ_2 lies inside u'. Let Δ_2 be $(\lambda' x' \cdot u'')v''$ (or resp. $\mathsf{box}\ u''\ \mathsf{with}\ v'_1, \ldots, v'_n\ \mathsf{for}\ x'_1, \ldots, x'_n$, with $x' = x'_j$ and $v'' = v'_j$ for some arbitrary j, $1 \leq j \leq n$). Then contracting Δ_1 replaces Δ_2 by $(\lambda' x' \cdot u''[v'/x])(v''[v'/x])$ (resp. $\mathsf{box}\ u''\ \mathsf{with}\ v'_1[v'/x], \ldots, v'_n[v'/x]\ \mathsf{for}\ x'_1, \ldots, x'_n$).
  If x is a proxy (meaning that Δ_1 must be a box-term), then $W_v(v''[v'/x]) = W_u(v'')$ by definition of the weight (the weight of the proxy is the weight of the term which is subtituted 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 $W(v''[v'/x]) \leq W(v'')$. In any case, $W(v''[v'/x]) \leq W(v'')$, and because the weighting of u is decreasing, for every occurrence x'^k of x' in $u''[v'/x]$ (equivalently, in u''), $k > W(v'') \geq W(v''[v'/x])$, so the weighting of v remains decreasing.

□

Lemma 5.4 *In λ_{S4}', every sequence of* (unbox) *steps is finite.*

Proof: Define the following translation from λ_{S4}' (without the commutative conversion rule) to the λ'-calculus, that is, the language built on variables, applications, λ-abstractions and λ'-redexes, with (β') as sole reduction rule:

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

If $u \to v$ by application of the (unbox) rule, then $I(u) \to^+ I(v)$ in λ' (by $n+1$ applications of rule (β')). As all developments in the λ-calculus are finite, all reductions in λ' are terminating, so every sequence of (unbox) steps in λ_{S4}' (without the commutative conversion rule) must be finite.

If $u_1 \sim u'_1 \to u_2 \sim u'_2 \to \ldots \to u_k$ is any sequence of (unbox) steps in λ_{S4}' (with the commutative conversion rule), notice that we can always postpone the application of the commutative conversion steps, so as to obtain a reduction sequence $u''_1 \to u''_2 \to \ldots \to u''_k$ of the same length, where for each i, $1 \leq i \leq k$, $u''_i \sim u_i$ (and in fact $u''_1 = u_1$). The latter is a sequence of (unbox) steps in the calculus without the commutative conversion rules, so its length is bounded by some function $k(u_1)$. 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 λ_{S4} are finite, that is:

Theorem 5.5 *The notion of reduction in λ_{S4}' is strongly normalizing.*

Proof: Let u_0 be a λ_{S4}'-term. By Lemma 5.2, there is a $\lambda_{S4}'^*$-term u such that $E_0(u) = u_0$ and the weighting of u is decreasing. Consider an arbitrary reduction in λ_{S4}' starting from u_0, and lift it to a corresponding reduction in $\lambda_{S4}'^*$ starting from u (replacing rules (box) and (ctract) by (box') and (ctract') respectively). For each term v in the sequence, let $W'(v)$ be the couple $(W(v), \nu(v))$, where $\nu(v)$ is the greatest length of a sequence of (unbox) steps that can be applied from $E_0(v)$ (which is finite by Lemma 5.4). By Lemma 5.3, $W'(v)$ decreases strictly in the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$, which is well-founded; therefore the reduction must be finite. □

Lemma 5.6 *The λ_{S4}'-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):

- Between (unbox) and (box): assume that $v_i = \mathsf{box}\ v\ \mathsf{with}\ w_1, \ldots, w_m\ \mathsf{for}\ y_1, \ldots, y_m$, then:

$$\mathsf{unbox}\ (\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n)$$

  reduces in one step to:

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

  by (unbox), but also to:

$$\begin{array}{ll}\mathsf{unbox}\ (& \mathsf{box}\ u[(\mathsf{box}\ v\ \mathsf{with}\ z_1, \ldots, z_m\ \mathsf{for}\ y_1, \ldots, y_m)/x_i] \\ & \mathsf{with}\ v_1, \ldots, v_{i-1}, w_1, \ldots, w_m, v_{i+1}, \ldots, v_n \\ & \mathsf{for}\ x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n)\end{array}$$

  by (box). By one application of (unbox), the latter reduces to the former, so the confluence diagram closes.

- Between (unbox) and (gc): assume that x_i is not free in u, then:

$$\mathsf{unbox}\ (\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n)$$

  reduces in one step to:

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

  by (unbox), but also to:

$$\mathsf{unbox}\ (\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$

  by (gc). But the latter reduces in one step to:

$$u[v_1/x_1, \ldots, v_{i-1}/x_{i-1}, v_{i+1}/x_{i+1}, \ldots, v_n/x_n]$$

  which is equal to the former, since x_i is not free in u.

- Between (unbox) and (ctract): when $v_i = v_j$ for $i \neq j$,

$$\mathsf{unbox}\ (\mathsf{box}\ u\ \mathsf{with}\ \ldots, v_i, \ldots, v_j, \ldots\ \mathsf{for}\ \ldots, x_i, \ldots, x_j, \ldots)$$

  reduces in one step either to:

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

  or to:

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

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

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

$$v_i = \mathsf{box}\ v'\ \mathsf{with}\ w'_1, \ldots, w'_{m'}\ \mathsf{for}\ x'_1, \ldots, x'_{m'}$$

  and

$$v_j = \mathsf{box}\ v''\ \mathsf{with}\ w''_1, \ldots, w''_{m''}\ \mathsf{for}\ x''_1, \ldots, x''_{m''}$$

  then

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots, v_i, \ldots, v_j, \ldots\ \mathsf{for}\ \ldots, x_i, \ldots, x_j, \ldots$$

  reduces in one step either to:

$$\begin{array}{l}\mathsf{box}\ u[(\mathsf{box}\ v'\ \mathsf{with}\ z'_1, \ldots, z'_{m'}\ \mathsf{for}\ x'_1, \ldots, x'_{m'})/x_i] \\ \mathsf{with}\ \ldots, w'_1, \ldots, w'_{m'}, \ldots, v_j, \ldots \\ \mathsf{for}\ \ldots, z'_1, \ldots, z'_{m'}, \ldots, x_j, \ldots\end{array}$$

or to:

$$\begin{array}{l}\textsf{box}\ u[(\textsf{box}\ v''\ \textsf{with}\ z''_1,\ldots,z''_{m''}\ \textsf{for}\ x''_1,\ldots,x''_{m''})/x_j]\\ \textsf{with}\ \ldots,v_i,\ldots,w''_1,\ldots,w''_{m''},\ldots\\ \textsf{for}\ \ldots,x_i,\ldots,z''_1,\ldots,z''_{m''},\ldots\end{array}$$

Both reduce by one application of box to:

$$\begin{array}{l}\textsf{box}\ u[(\textsf{box}\ v'\ \textsf{with}\ z'_1,\ldots,z'_{m'}\ \textsf{for}\ x'_1,\ldots,x'_{m'})/x_i,(\textsf{box}\ v''\ \textsf{with}\ z''_1,\ldots,z''_{m''}\ \textsf{for}\ x''_1,\ldots,x''_{m''})/x_j]\\ \textsf{with}\ \ldots,w'_1,\ldots,w'_{m'},\ldots,w''_1,\ldots,w''_{m''},\ldots\\ \textsf{for}\ \ldots,z'_1,\ldots,z'_{m'},\ldots,z''_1,\ldots,z''_{m''},\ldots\end{array}$$

- Between (box) and (gc). If $i \neq j$,

$$v_i = \textsf{box}\ v\ \textsf{with}\ w_1,\ldots,w_m\ \textsf{for}\ x_1,\ldots,x_m$$

and x_j is not free in u, then:

$$\textsf{box}\ u\ \textsf{with}\ \ldots,v_i,\ldots,v_j,\ldots\ \textsf{for}\ \ldots,x_i,\ldots,x_j,\ldots$$

reduces in one (box) step either to:

$$\begin{array}{l}\textsf{box}\ u[(\textsf{box}\ v\ \textsf{with}\ z_1,\ldots,z_m\ \textsf{for}\ x_1,\ldots,x_m)/x_i]\\ \textsf{with}\ \ldots,w_1,\ldots,w_m,\ldots,v_j,\ldots\\ \textsf{for}\ \ldots,z_1,\ldots,z_m,\ldots,x_j,\ldots\end{array}$$

or to:

$$\textsf{box}\ u\ \textsf{with}\ \ldots,v_i,\ldots,\ldots\ \textsf{for}\ \ldots,x_i,\ldots,\ldots$$

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

$$\begin{array}{l}\textsf{box}\ u[(\textsf{box}\ v\ \textsf{with}\ z_1,\ldots,z_m\ \textsf{for}\ x_1,\ldots,x_m)/x_i]\\ \textsf{with}\ \ldots,w_1,\ldots,w_m,\ldots,\ldots\\ \textsf{for}\ \ldots,z_1,\ldots,z_m,\ldots,\ldots\end{array}$$

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

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

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

$$\textsf{box}\ u\ \textsf{with}\ \ldots,v_i,\ldots,v_j,\ldots\ \textsf{for}\ \ldots,x_i,\ldots,x_j,\ldots$$

where:

$$v_i = v_j = \textsf{box}\ v\ \textsf{with}\ w_1,\ldots,w_m\ \textsf{for}\ x_1,\ldots,x_m$$

contracts either to:

$$\begin{array}{l}\textsf{box}\ u[(\textsf{box}\ v\ \textsf{with}\ z_1,\ldots,z_m\ \textsf{for}\ x_1,\ldots,x_m)/x_i]\\ \textsf{with}\ \ldots,w_1,\ldots,w_m,\ldots,v_j,\ldots\\ \textsf{for}\ \ldots,z_1,\ldots,z_m,\ldots,x_j,\ldots\end{array}$$

by (box) or to:

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

by (ctract).

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

$$\begin{array}{l}\mathsf{box}\ u[(\mathsf{box}\ v\ \mathsf{with}\ z_1,\ldots,z_m\ \mathsf{for}\ x_1,\ldots,x_m)/x_i,(\mathsf{box}\ v\ \mathsf{with}\ z'_1,\ldots,z'_m\ \mathsf{for}\ x_1,\ldots,x_m)/x_j]\\ \mathsf{with}\ \ldots,w_1,\ldots,w_m,\ldots,w_1,\ldots,w_m,\ldots\\ \mathsf{for}\ \ldots,z_1,\ldots,z_m,\ldots,z'_1,\ldots,z'_m,\ldots\end{array}$$

then by m applications of (ctract) to:

$$\begin{array}{l}\mathsf{box}\ u[(\mathsf{box}\ v\ \mathsf{with}\ z_1,\ldots,z_m\ \mathsf{for}\ x_1,\ldots,x_m)/x_i,(\mathsf{box}\ v\ \mathsf{with}\ z_1,\ldots,z_m\ \mathsf{for}\ x_1,\ldots,x_m)/x_j]\\ \mathsf{with}\ \ldots,w_1,\ldots,w_m,\ldots,,\ldots\\ \mathsf{for}\ \ldots,z_1,\ldots,z_m,\ldots,,\ldots\end{array}$$

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:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,v_i,\ldots,v_j,\ldots\ \mathsf{for}\ \ldots,x_i,\ldots,x_j,\ldots$$

reduces in one step either to:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,\ldots,v_j,\ldots\ \mathsf{for}\ \ldots,\ldots,x_j,\ldots$$

or to:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,v_i,\ldots,\ldots\ \mathsf{for}\ \ldots,x_i,\ldots,\ldots$$

which both reduce by (gc) to:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,\ldots,\ldots\ \mathsf{for}\ \ldots,\ldots,\ldots$$

- Between (gc) and (ctract): when $v_i = v_j$ and x_i is not free in u,

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,v_i,\ldots,v_j,\ldots\ \mathsf{for}\ \ldots,x_i,\ldots,x_j,\ldots$$

reduces in one (gc) step to:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,\ldots,v_j,\ldots\ \mathsf{for}\ \ldots,\ldots,x_j,\ldots$$

or to:

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

Now, since x_i is not free in u, the latter is an α-renamed version of:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,v_i,\ldots,\ldots\ \mathsf{for}\ \ldots,x_j,\ldots,\ldots$$

This, in turn, is just:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,v_j,\ldots,\ldots\ \mathsf{for}\ \ldots,x_j,\ldots,\ldots$$

since $v_i = v_j$, so it is $\sim$-equivalent to:

$$\mathsf{box}\ u\ \mathsf{with}\ \ldots,\ldots,v_j,\ldots\ \mathsf{for}\ \ldots,\ldots,x_j,\ldots$$

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 (ctract) and itself: given $v_i = v_j$ and $v_k = v_l$, with $i \neq j$, $k \neq l$ and $(i,j) \neq (k,l)$,

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

reduces in one step either to the box of $u[x_i/x_j]$ with all bindings but the jth, or to the box of $u[x_k/x_l]$ with all bindings but the lth.

If $j = l$, then $i \neq k$ by assumption, and $v_j = v_l$, so that these two terms are in fact $\sim$-equivalent.

If $j \neq l$, then the former reduces by (ctract) to the box of $u[x_i/x_j][x_k/x_l]$ with all bindings but the jth and the lth, and the latter reduces by (ctract) to the box of $u[x_k/x_l][x_i/x_j]$ with all bindings but the jth and the lth. Since $j \neq l$, these are the same terms.

□

Theorem 5.7 *The λ_{S4}-calculus is confluent.*

Proof: Let $\xrightarrow{1}$ be the relation defined as: $u \xrightarrow{1} v$ if and only if there is a ${\lambda_{S4}}'$-term u' such that $u' \longrightarrow^* v$ in ${\lambda_{S4}}'$, and $E(u') = u$.

We claim that $\xrightarrow{1}$ is confluent. Indeed, let $u \xrightarrow{1} u_1$ and $u \xrightarrow{1} u_2$. Then, by definition, there are two terms u'_1 and u'_2, which we obtain from u by adding primes on some λ-redexes (i.e., such that $E(u'_1) = E(u'_2) = u$), such that $u'_1 \longrightarrow^* u_1$ and $u'_2 \longrightarrow^* u_2$ in ${\lambda_{S4}}'$. Build u' from u by adding primes on λ'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 λ-redexes. By an easy induction on the length of derivations, $u'_1 \longrightarrow^* u''_1$ and $u'_2 \longrightarrow^* u''_2$ in ${\lambda_{S4}}'$, where $E(u''_1) = u_1$ and $E(u''_2) = u_2$. By Theorem 5.5, u''_1 and u''_2 have ${\lambda_{S4}}'$-normal forms; by Lemma 5.6, these normal forms are the same, call them v. Because v is normal, $E(v) = v$, that is, v is in fact a λ_{S4}-term, and clearly the reductions from u''_1 and u''_2 to v translate to λ_{S4}-reductions, by erasing all primes from redexes and replacing (β'), (box') and (ctract')-contractions by (β), (box) and (ctract)-contractions respectively. Therefore, $u_1 \xrightarrow{1} v$ and $u_2 \xrightarrow{1} v$.

Then, we claim that the reflexive transitive closure $\longrightarrow^*$ of the one-step λ_{S4}-reduction $\longrightarrow$ is equal to the reflexive transitive closure $\xrightarrow{1}^*$ of $\xrightarrow{1}$. Indeed, $\xrightarrow{1}$ is a sub-relation of $\longrightarrow^*$; and conversely, $\longrightarrow$ is a sub-relation of $\xrightarrow{1}$: if $u \longrightarrow v$ by some rule other than (β), this is trivial, otherwise just add a prime on the λ-redex that is contracted and (β')-contract.

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

Theorem 5.8 *Subject reduction holds in λ_{S4}.*

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

The deep reason why this works is that reduction rules in the typed fragment of λ_{S4} are really proof transformations, and that proofness is typedness. Since we have in fact derived the reduction rules from allowed proof transformations, we won't bother the reader by translating all this work into yet another language. □

Theorem 5.9 *All typed λ_{S4}-terms are strongly normalizing.*

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

$$D(\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n) = D(u)[D(v_1)/x_1, \ldots, D(v_n)/x_n]$$
$$D(\mathsf{unbox}\ u) = D(u)$$

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

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

Now, let $u_1 \longrightarrow u_2 \longrightarrow \ldots \longrightarrow u_i \longrightarrow \ldots$ be a reduction starting from the well-typed λ_{S4}-term u_1. Then $D(u_1) \xrightarrow{=} D(u_2) \xrightarrow{=} \ldots \xrightarrow{=} D(u_i) \xrightarrow{=} \ldots$ in the simply-typed λ-calculus, where $\xrightarrow{=}$ is the reflexive closure of reduction in this calculus; indeed, every contraction by (β) translates by D into a contraction by the same rule, and if $u_i \longrightarrow u_{i+1}$ by some other rule, then $D(u_i) = D(u_{i+1})$.

Since the simply-typed λ-calculus is strongly normalizing, there are only finitely many $\xrightarrow{=}$ steps that are not equalities in the above reduction. On the other hand, those steps that are equalities are erasings of $(\mathsf{unbox}\ )$, (box), (gc) or (ctract) steps, which are in fact steps in ${\lambda_{S4}}'$; by Theorem 5.5, there can be only finitely many consecutive $\xrightarrow{=}$ steps that are equalities. Therefore the original λ_{S4}-reduction is finite. □

Definition 5.5 *We say that a λ_{S4}-term is an* elimination *if and only if it is of the form vw or* unbox v*.*

Let u be a λ_{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 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 vw is the sum of those of v and w 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. $\square$

Lemma 5.11 *Let u be a well-typed normal elimination. Then $d_1(u)$ is defined, and is of the form xw or* 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 unbox u_n.

Then, u_n cannot be an abstraction, by typing, or a quotation box _ with _ for _ by normality (rule (unbox) does not apply). So u_n must be a variable x. $\square$

Theorem 5.12 *Every typable normal λ_{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 λ-abstraction, we complete the proofs obtained by induction hypothesis by $(\Rightarrow R)$.

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 (gc) 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 $(\square L)$, as 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, ..., v_n are variables. Because u is normal, it is not a (ctract) redex, so the variables v_1, ..., v_n are pairwise distinct. By induction hypothesis, $u'[v_1/x_1, \ldots, v_n/x_n]$ (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 $(\square R)$ below, as follows:

$$\begin{array}{c} \pi(u'[v_1/x_1, \ldots, v_n/x_n]) \\ \vdots \\ (\square R)\ \dfrac{v_1 : \square\Phi_1, \ldots, v_n : \square\Phi_n \vdash u'[v_1/x_1, \ldots, v_n/x_n] : \Phi}{v_1 : \square\Phi_1, \ldots, v_n : \square\Phi_n \vdash u : \square\Phi} \end{array}$$

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

Theorem 5.13 *Every typable λ_{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.6, by a modification of Hindley's type inferencing algorithm. Typing $\mathsf{unbox}\ u$ is done as we were typing ›u, and computing $T(, , \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n)$ follows the same lines as for ‘u: compute $(\sigma_1, \Phi_1) = T(, , v_1)$, $(\sigma_2, \Phi_2) = T(, \sigma_1, v_2)$, …, $(\sigma_n, \Phi_n) = T(, \sigma_1 \ldots \sigma_{n-1}, v_2)$; let Φ'_1 be $\Phi_1 \sigma_2 \sigma_3 \ldots \sigma_n$, Φ'_2 be $\Phi_2 \sigma_3 \ldots \sigma_n$, …, Φ'_n be Φ_n; create n fresh type variables α_1, …, α_n, and compute the most general common substitution σ of Φ'_i with $\Box\alpha_i$, $1 \leq i \leq n$; then return $T((x_1 : \Box\alpha_1, \ldots, x_n : \Box\alpha_n)\sigma, u)$. (If any operation fails, we fail.) The theorem follows from the same arguments as in Theorem B.6. □

5.2 Why λ_{S4} is Not Satisfactory

We have said that this system was almost perfect. Unfortunately, it has the following defect. The commuting conversion above amount to considering the with/for part of a box operator as a multiset $\{x_i \mapsto v_i \mid 1 \leq i \leq n\}$ of bindings, not just an ordered list of bindings $x_1 := v_1$, …, $x_n := v_n$. And the commuting conversion is necessary to equate proofs 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 (ctract) is needed to this purpose, and this forces us to understand the with/for parts as *sets* of bindings (in fact, as maps).

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

$$
\begin{array}{rc}
(Ax) & \overline{x : \Box\Phi, x_0 : \Psi \vdash x_0 : \Psi} \quad (Ax)\ \overline{x : \Box\Phi \vdash x : \Box\Phi} \\
(\Rightarrow L) & \overline{\quad x : \Box\Phi, x_1 : \Box\Phi \Rightarrow \Psi \vdash x_1 x : \Psi \quad} \quad (Ax)\ \overline{x : \Box\Phi \vdash x : \Box\Phi} \\
(\Rightarrow L) & \overline{\qquad x : \Box\Phi, u : \Box\Phi \Rightarrow \Box\Phi \Rightarrow \Psi \vdash uxx : \Psi \qquad} \\
(\Rightarrow R) & \overline{\qquad x : \Box\Phi \vdash \lambda u \cdot uxx : (\Box\Phi \Rightarrow \Box\Phi \Rightarrow \Psi) \Rightarrow \Psi \qquad} \\
(\Box R) & \overline{\quad x : \Box\Phi \vdash \mathsf{box}\ \lambda u \cdot uzz\ \mathsf{with}\ x\ \mathsf{for}\ z : \Box((\Box\Phi \Rightarrow \Box\Phi \Rightarrow \Psi) \Rightarrow \Psi) \quad}
\end{array}
$$

We cannot dispense with the commuting conversion either, otherwise we would lose confluence. For example, when $v_2 = v_4$ is not boxed, (writing (u, v) instead of the more cumbersome $\lambda z \cdot zuv$),

$$\mathsf{box}\ ((x_1, x_3), (x_4, x_5))\ \mathsf{with}\ v_1, v_2, v_3, v_4, v_5\ \mathsf{for}\ x_1, x_2, x_3, x_4, x_5$$

rewrites by rule (gc) to

$$\mathsf{box}\ ((x_1, x_3), (x_4, x_5))\ \mathsf{with}\ v_1, v_3, v_4, v_5\ \mathsf{for}\ x_1, x_3, x_4, x_5$$

and by rule (ctract) to

$$\mathsf{box}\ ((x_1, x_3), (x_2, x_5))\ \mathsf{with}\ v_1, v_2, v_3, v_5\ \mathsf{for}\ x_1, x_2, x_3, x_5$$

If v_1, v_2, v_3 and v_5 are distinct variables, then these two terms are normal and not α-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, not 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 λ_{S4}-calculus is not a computationally adequate formalism for describing how cut-elimination works in S4. This is a hint that instead of *explaining* the computation contents of cut-elimination, λ_{S4} in fact hides it under the carpet.

Another problem in the λ_{S4}-calculus is related to the rules (gc) and (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, that is, new clauses in the definition of the commuting conversion relation $\sim$. On the other hand, it is more natural to see them as rewrite rules from a computational viewpoint; this is a slight, but annoying mismatch. To balance this, we define:

Definition 5.6 ($\lambda^{\approx}_{S4}$) *The $\lambda^{\approx}_{S4}$-calculus is defined as follows.*

The $\lambda^{\approx}_{S4}$-terms are all equivalence classes of λ_{S4}-terms modulo $\approx$, where $\approx$ is the smallest congruence containing $\sim$, (gc) *and* (ctract)*. (That is, such that whenever $u \sim v$, or $u \to v$ by the latter rules, then $u \approx v$.)*

The rewrite rules are rules (β), $(\mathsf{unbox}\ )$ *and* (box).

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

$$\frac{u \approx v \quad , \vdash u : \Phi}{, \vdash v : \Phi}$$

We need the additional typing rule, because rule $(\Box\mathcal{I})$ is not invariant under $\approx$. This has the unfortunate consequence that a term might be typable but still have untypable subterms (consider $\mathsf{box}\ x\ \mathsf{with}\ u, v\ \mathsf{for}\ x, y$, with u typable but v untypable). The typed calculus also exhibits non-terminating behaviours: consider for example $\mathsf{box}\ x\ \mathsf{with}\ u, \Omega\ \mathsf{for}\ x, y$, where Ω is an (untypable) term with an infinite rewrite $\Omega \to \Omega_1 \to \Omega_2 \to \ldots$, then $\mathsf{box}\ x\ \mathsf{with}\ u, \Omega\ \mathsf{for}\ x, y \to \mathsf{box}\ x\ \mathsf{with}\ u, \Omega_1\ \mathsf{for}\ x, y \to \mathsf{box}\ x\ \mathsf{with}\ u, \Omega_2\ \mathsf{for}\ x, y \to \ldots$ 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 is, terms where variables are annotated with their types, à la Church:

Definition 5.7 *The set of* typed λ_{S4} pre-terms *is described by the same grammar as that of the* λ_{S4}*-terms, except that each variable has a given type* Φ*. If the type of* x *is* Φ*, 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 $x : \Phi$*,* Φ *is the type of* x*.*

The typed λ_{S4}-terms *are all typable pre-terms. The* $\lambda_{\mathrm{S4}}^{\approx}$-terms *are all classes of typed* λ_{S4}*-terms modulo* $\approx$*.*

Then, every sequence of rewriting steps of typed λ_{S4}-terms terminates, but there are still infinite rewriting sequences in $\lambda_{\mathrm{S4}}^{\approx}$. A sequence of rewriting steps in $\lambda_{\mathrm{S4}}^{\approx}$ is indeed a sequence of terms such that $u_0 \approx u'_0 \longrightarrow u_1 \approx u'_1 \longrightarrow \ldots \longrightarrow u_n \approx u'_n \ldots$. We then have, for instance, the looping rewriting sequence $\mathsf{box}\ x\ \mathsf{with}\ y\ \mathsf{for}\ x \approx \mathsf{box}\ x\ \mathsf{with}\ y, (\lambda z \cdot z)u\ \mathsf{for}\ x, x' \longrightarrow \mathsf{box}\ x\ \mathsf{with}\ y, u\ \mathsf{for}\ x, x' \approx \mathsf{box}\ x\ \mathsf{with}\ y\ \mathsf{for}\ x$, which can be repeated infinitely many times.

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

We still have a dilemma with (gc) and (ctract). The natural notion of proof in S4 is given by *free-form* natural deductions, 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 (ctract) (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 equivalence classes modulo $\approx$, but rather equivalence classes of (gc), (ctract)-normal terms modulo $\sim$. 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 $\lambda_{\mathrm{S4}}^{\approx}$-terms (equivalently, (gc), (ctract)-normal λ_{S4}-terms, which are unique representatives of $\lambda_{\mathrm{S4}}^{\approx}$-terms) are in bijection with free-form natural deduction proofs (equivalently, natural deduction proofs in sequent form, modulo weakenings), and where derivations following the above 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 $\overline{\mathbf{S4}}$ *(resp.* $\underline{\mathbf{S4}}$*) is defined as follows.*

The objects are all typed $\lambda_{\mathrm{S4}}^{\approx}$*-terms. The morphisms are all derivations between* $\lambda_{\mathrm{S4}}^{\approx}$*-terms (resp. all derivations* $u_0 \approx u'_0 \longrightarrow u_1 \approx u'_1 \longrightarrow \ldots \longrightarrow u_n \approx u'_n$*, where each* $\longrightarrow$*-step is a* (β)*,* $(\mathsf{unbox}\ )$ *or* (box) *step, and where for each* i*,* $0 \leq i \leq n$*,* u'_i *is the* (gc)*,* (ctract)*-normal form of* u_i*).*

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

Both $\overline{\mathbf{S4}}$ and $\underline{\mathbf{S4}}$ have the same problem as the λ_{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 λ_{S4}: we take it as an indication that a lot was hidden in the rules of λ_{S4}.

6 Related Works

The search for functional interpretations of modal logics was initiated by Dov Gabbay and Ruy de Queiroz [GdQ90], who were interested in various non-classical logics. Their interpretation of modal logics (not only S4) is a direct recoding of the notion of Kripke models inside the proof structure. (See pp.38–39, op.cit.) They introduce world variables w, write $\forall w \cdot \Phi(w)$ instead of $\Box\Phi$, 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 $w : \mathcal{W}$ (w is a world in the chosen universe $\mathcal{W}$), we can derive $u(w) : \Phi(w)$, then we can discharge the assumption $w : \mathcal{W}$ and derive $\Lambda w \cdot u(w) : \Box\Phi$. (Discharging the assumption, 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 hard. Finding the correct reduction rules is trickier, and 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, rewrite the S4 $\Box$ into the linear ! modality ("of course"); then $(\Box I)$ is promotion, $(\Box E)$ is dereliction, and all other rules of linear logic can be proved from the rules of S4, if we overlook the fact that linear sequents use *sequences* of formulas, not sets of formulas, as contexts.

This led Gavin Bierman and Valeria de Paiva to adapt a functional interpretation of intuitionistic linear logic to S4 [BdP92, BdP95]. They inferred that the right categories of natural deduction S4 proofs were the cartesian closed categories with coproducts and a monoidal comonad. The notion of reduction they propose is however incomplete, in so far as there may remain cuts in proofs corresponding to normal terms, i.e. execution of a program may deadlock before reaching a value. We have corrected this 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, to try to understand how S4 works, in the hope of understanding more about linear logic. They define a calculus, named $\lambda^{\to\Box}$, of constructions for S4. They try to solve the problem of the $(\Box R)$ rule by using sequents of the form $,_1; ,_2; \ldots ; ,_n \vdash \Phi$, where $,_1; ,_2; \ldots ; ,_n$ is a *context stack*, i.e. a list of contexts $,_i$, $1 \leq i \leq n$. This is a good start, because the main problem in the $(\Box R)$ rule is that, when going from the conclusions to the premises, we lose all unboxed formulas on the left-hand side (in $,'$), which in turn prevents cuts from being pushed upwards. Instead of dropping $,'$, 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 $(\Box R)$ by the following two rules:

$$\Box I\ \frac{,_1; \ldots ; ,_n; \cdot \vdash u : \Phi}{,_1; \ldots ; ,_n \vdash \text{‘}u : \Box\Phi} \quad pop\ \frac{,_1; \ldots ; ,_n; ,_{n+1} \vdash u : \Box\Phi}{,_1; \ldots ; ,_n \vdash u : \Box\Phi}$$

where $\cdot$ is the empty context, and where the authors use the word "box" instead of ‘, and "unbox" instead of ’. This system is already closer to the one that we shall develop in Part II, in that we can interpret their sequents $,_1; \ldots ; ,_n \vdash u : \Phi$ as the internalized and currified version $,_1 \vdash \Psi_2 \overset{\Box}{\Rightarrow} \ldots \Psi_n \overset{\Box}{\Rightarrow} \Phi$, where for each i, $2 \leq i \leq n$, Ψ_i is defined as $\Psi_i^1 \times \ldots \times \Psi_i^{m_i} \times \top$ if $,_i$ is $\Psi_i^1, \ldots, \Psi_i^{m_i}$. And conversely, there is a sublanguage of our formulas that we can interpret as sequents with context stacks.

Their system, however, has several defects. First, although typing is decidable, it is an NP-complete problem. This is due to the fact that *pop* is a parasitic rule, which we can guess from the fact that it has no computational effect at all. This was corrected in [DP95], where the authors also note that the box and unbox operators naturally represent an eval/quote-like mechanism.

Second, and more seriously, 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 $(\Box R)$ 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 λ_{S4}-calculus, as well as the $\lambda\mathtt{ev}Q$-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 λ_{S4} 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 $, \vdash \Phi$ if Φ can be deduced from the set of assumptions $,$ in the Hilbert-style system, and $\vdash , \vdash \Phi$ if the sequent $, \vdash \Phi$ is deductible in the Gentzen-style sequent system.

We prove the following:

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

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

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

We show how to transform a Hilbert-style proof of $, \vdash \Phi$ into a sequent proof of $\Box, \vdash \Phi$. Instead of just producing a proof, actually, we shall leave the constructions in, so 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 Φ is a member of $,$, then we translate this to the following proof:

$$(\Box L)\,\dfrac{(Ax)\,\dfrac{}{\Box, , x' : \Phi \vdash x' : \Phi}}{\Box, , x : \Box\Phi \vdash {,}x : \Phi}$$

- if Φ is an instance of one of the axiom schemas:
  (s) then we produce the following proof:

$$(\Rightarrow R)\,\dfrac{(\Rightarrow R)\,\dfrac{(\Rightarrow R)\,\dfrac{(\Rightarrow L)\,\dfrac{(\Rightarrow L)\,\dfrac{(\Rightarrow L)\,\dfrac{(Ax)\,\dfrac{}{\begin{array}{c}, ', x_3 : \Phi_3,\\ x_2 : \Phi_2,\\ z : \Phi_1\\ \vdash x_3 : \Phi_3\end{array}} \quad (Ax)\,\dfrac{}{\begin{array}{c}, ',\\ x_2 : \Phi_2,\\ z : \Phi_1\\ \vdash x_2 : \Phi_2\end{array}}}{\begin{array}{c}, ', x_1 : \Phi_2 \Rightarrow \Phi_3,\\ x_2 : \Phi_2, z : \Phi_1\\ \vdash x_1 x_2 : \Phi_3\end{array}} \quad (Ax)\,\dfrac{}{\begin{array}{c}, ',\\ x_2 : \Phi_2,\\ z : \Phi_1\\ \vdash z : \Phi_1\end{array}}}{\begin{array}{c}, ', x : \Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3,\\ x_2 : \Phi_2, z : \Phi_1\\ \vdash xzx_2 : \Phi_3\end{array}} \quad (Ax)\,\dfrac{}{\begin{array}{c}, ',\\ x : \Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3,\\ z : \Phi_1\\ \vdash z : \Phi_1\end{array}}}{\begin{array}{c}, ', x : \Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3, y : \Phi_1 \Rightarrow \Phi_2, z : \Phi_1\\ \vdash xz(yz) : \Phi_3\end{array}}}{\begin{array}{c}, ', x : \Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3, y : \Phi_1 \Rightarrow \Phi_2\\ \vdash \lambda z \cdot xz(yz) : \Phi_1 \Rightarrow \Phi_3\end{array}}}{\begin{array}{c}, ', x : \Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3\\ \vdash \lambda y \cdot \lambda z \cdot xz(yz) : (\Phi_1 \Rightarrow \Phi_2) \Rightarrow (\Phi_1 \Rightarrow \Phi_3)\end{array}}}{, ' \vdash \lambda x \cdot \lambda y \cdot \lambda z \cdot xz(yz) : (\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_3) \Rightarrow (\Phi_1 \Rightarrow \Phi_2) \Rightarrow (\Phi_1 \Rightarrow \Phi_3)}$$

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

  (k) then we build:

$$(\Rightarrow R)\,\dfrac{(\Rightarrow R)\,\dfrac{(Ax)\,\dfrac{}{, ', x : \Phi_1, y : \Phi_2 \vdash x : \Phi_1}}{, ', x : \Phi_1 \vdash \lambda y \cdot x : \Phi_2 \Rightarrow \Phi_1}}{, ', \vdash \lambda x \cdot \lambda y \cdot x : \Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_1}$$

  where $, ' = \Box,$.

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

$$(\Rightarrow R)\,\dfrac{(\Rightarrow R)\,\dfrac{(\Box R)\,\dfrac{(\Box L)\,\dfrac{(\Box L)\,\dfrac{(\Rightarrow L)\,\dfrac{(Ax)\,\dfrac{}{x_2 : \Phi_2, x_1 : \Phi_1 \vdash x_2 : \Phi_2} \quad (Ax)\,\dfrac{}{x_1 : \Phi_1 \vdash x_1 : \Phi_1}}{x_3 : \Phi_1 \Rightarrow \Phi_2, x_1 : \Phi_1 \vdash x_3 x_1 : \Phi_2}}{x : \Box(\Phi_1 \Rightarrow \Phi_2), x_1 : \Phi_1 \vdash ({,}x)x_1 : \Phi_2}}{x : \Box(\Phi_1 \Rightarrow \Phi_2), y : \Box\Phi_1 \vdash ({,}x)({,}y) : \Phi_2}}{\Box, , x : \Box(\Phi_1 \Rightarrow \Phi_2), y : \Box\Phi_1 \vdash (({,}x)({,}y))^{\text{‘}} : \Box\Phi_2}}{\Box, , x : \Box(\Phi_1 \Rightarrow \Phi_2) \vdash \lambda y \cdot (({,}x)({,}y))^{\text{‘}} : \Box\Phi_1 \Rightarrow \Box\Phi_2}}{\Box, \vdash \lambda x \cdot \lambda y \cdot (({,}x)({,}y))^{\text{‘}} : \Box(\Phi_1 \Rightarrow \Phi_2) \Rightarrow \Box\Phi_1 \Rightarrow \Box\Phi_2}$$

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

(T)

$$\begin{array}{lc} (Ax) & \overline{\;,',x':\Phi\vdash x':\Phi\;} \\ (\Box L) & \overline{\;,',x:\Box\Phi\vdash \text{ˏ}x:\Phi\;} \\ (\Rightarrow R) & \overline{\;,'\vdash\lambda x\cdot \text{ˏ}x:\Box\Phi\Rightarrow\Phi\;}\end{array}$$

where $,'$ is $\Box ,$.

(4) , produce:

$$\begin{array}{lc} (Ax) & \overline{\;\Box ,\ ,x:\Box\Phi\vdash x:\Box\Phi\;} \\ (\Box R) & \overline{\;\Box ,\ ,x:\Box\Phi\vdash x^{\text{‘}}:\Box\Box\Phi\;} \\ (\Rightarrow R) & \overline{\;\Box ,\ \vdash\lambda x\cdot x^{\text{‘}}:\Box\Phi\Rightarrow\Box\Box\Phi\;}\end{array}$$

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

Now, assume that we have a Hilbert-style proof of $, \vdash \Phi$, and that all Hilbert-style proofs of strictly shorter length translate to sequent proofs. We prove that the same happens with $, \vdash \Phi$:

- if Φ was obtained by modus ponens (rule (MP)), i.e. we have strictly shorter proofs of $, \vdash \Phi_1 \Rightarrow \Phi$ and of $, \vdash \Phi_1$, then we get derivations (π) of $,' \vdash u : \Phi_1 \Rightarrow \Phi$ and (π') of $,' \vdash v : \Phi_1$ (where $,' = \Box ,$). Therefore, the following:

$$\dfrac{\dfrac{\begin{matrix}(\pi)\\ \vdots\\ ,'\vdash u:\Phi_1\Rightarrow\Phi\end{matrix}\quad \dfrac{(Ax)\ \overline{,',x_2:\Phi,x_1:\Phi_1\vdash x_2:\Phi}\quad (Ax)\ \overline{,',x_1:\Phi_1\vdash x_1:\Phi_1}}{,',x_3:\Phi_1\Rightarrow\Phi,x_1:\Phi_1\vdash x_3x_1:\Phi}(\Rightarrow L)}{,',x_1:\Phi_1\vdash ux_1:\Phi}(Cut)\quad \begin{matrix}(\pi')\\ \vdots\\ ,'\vdash v:\Phi_1\end{matrix}}{,'\vdash uv:\Phi}(Cut)$$

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

- if Φ is obtained by the necessitation rule (Nec), then Φ is of the form $\Box\Phi'$, where $, \vdash \Phi'$ has a stricly shorter proof. The latter translates to a derivation (π) of $\Box , \vdash u : \Phi'$. Then:

$$\begin{array}{lc} & (\pi) \\ & \vdots \\ & \Box ,\ \vdash u:\Phi' \\ (\Box R) & \overline{\;\Box ,\ \vdash u^{\text{‘}}:\Box\Phi'\;}\end{array}$$

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

QED.

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

We now show that if $\Box , \vdash \Phi$ is derivable in the sequent system, then $, \vdash \Phi$ 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 $, , \Phi_1 \vdash \Phi_2$ into proofs of $, \vdash \Box\Phi_1 \Rightarrow \Phi_2$.

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

Proof: If $\Phi_1 \Rightarrow \Phi_2$ is provable from , , then $\Phi_1 \Rightarrow \Phi_2$ is also provable from $, , \Phi_1$. Moreover, Φ_1 is provable from $, , \Phi_1$, and by modus ponens, Φ_2 is provable from $, , \Phi_1$.

Conversely, assume that Φ_2 is provable from $, , \Phi_1$. We build a proof of $, \vdash \Phi_1 \Rightarrow \Phi_2$ by induction on the length of the proof of $, , \Phi_1 \vdash \Phi_2$:

- if Φ_2 is Φ_1, we build a proof of $\Phi_1 \Rightarrow \Phi_1$: we can first prove $(\Phi_1 \Rightarrow (\Phi_2 \Rightarrow \Phi_1) \Rightarrow \Phi_1) \Rightarrow (\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_1) \Rightarrow (\Phi_1 \Rightarrow \Phi_1)$ by instantiating (s) (replace Φ_3 by Φ_1, and Φ_2 by $\Phi_2 \Rightarrow \Phi_1$); then, we can prove $(\Phi_1 \Rightarrow (\Phi_2 \Rightarrow \Phi_1) \Rightarrow \Phi_1)$ by instantiating (k) (replace Φ_2 by $\Phi_2 \Rightarrow \Phi_1$). By modus ponens, we infer $(\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_1) \Rightarrow (\Phi_1 \Rightarrow \Phi_1)$. But then $(\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_1)$ is another instance of (k), so by modus ponens again, we infer $\Phi_1 \Rightarrow \Phi_1$, i.e. $\Phi_1 \Rightarrow \Phi_2$. (Notice the analogy with the translation from the λ-term $\lambda x \cdot x$ to the combinatory term SKK.)
- if Φ_2 is a formula in , , then we can first prove $\Phi_2 \Rightarrow \Phi_1 \Rightarrow \Phi_2$, which is an instance of (k), then apply modus ponens on this and Φ_2 to get $\Phi_1 \Rightarrow \Phi_2$. (Notice the analogy with the translation from $\lambda x \cdot y$ to Ky.)
- if Φ_2 was deduced by modus ponens, i.e. there is a formula Φ_3 which is provable from , , Φ_1, and such that $\Phi_3 \Rightarrow \Phi_2$ is provable from , , Φ_1. Then by induction hypothesis, $\Phi_1 \Rightarrow \Phi_3 \Rightarrow \Phi_2$ and $\Phi_1 \Rightarrow \Phi_3$ are provable from , . Then, we produce $(\Phi_1 \Rightarrow \Phi_3 \Rightarrow \Phi_2) \Rightarrow (\Phi_1 \Rightarrow \Phi_3) \Rightarrow (\Phi_1 \Rightarrow \Phi_2)$, 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 $\lambda x \cdot tt'$ to $S(\lambda x \cdot t)(\lambda x \cdot t')$.)
- if Φ_2 was an instance of (s) or (k), then we produce $\Phi_2 \Rightarrow \Phi_1 \Rightarrow \Phi_2$, which is an instance of (k), then we apply modus ponens to get $\Phi_1 \Rightarrow \Phi_2$.

□

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

Proof: If $\Box\Phi_1 \Rightarrow \Phi_2$ is provable from , , then $\Box\Phi_1 \Rightarrow \Phi_2$ is also provable from , , Φ_1. Moreover, $\Box\Phi_1$ is provable from , , Φ_1 by necessitation, then by modus ponens, Φ_2 is provable from , , Φ_1.

Converserly, assume that Φ_2 is provable from , , Φ_1. We build a proof of , $\vdash \Box\Phi_1 \Rightarrow \Phi_2$ by induction on the length of the proof of , , $\Phi_1 \vdash \Phi_2$, as in the proof of Lemma A.2. In all cases of the latter lemma, then we notice that , , $\Box\Phi_1 \vdash \Phi_2$ was also provable by using (T) and modus ponens, and then we replace all uses of Φ_1 by $\Box\Phi_1$ in the latter proof. This handles the case where Φ_2 was obtained by any rule other than necessitation or by instantiating the non-modal axioms.

If Φ_2 was obtained by necessitation, i.e. Φ_2 is $\Box\Phi_3$, where , , $\Phi_1 \vdash \Phi_3$ is provable, then by induction hypothesis we can deduce $\Box\Phi_1 \Rightarrow \Phi_3$ from , . Then, by necessitation, we infer $\Box(\Box\Phi_1 \Rightarrow \Phi_3)$. Then, we produce $\Box(\Box\Phi_1 \Rightarrow \Phi_3) \Rightarrow \Box\Box\Phi_1 \Rightarrow \Box\Phi_3$, which is an instance of (K). By modus ponens, we get $\Box\Box\Phi_1 \Rightarrow \Box\Phi_3$. Produce $(\Box\Box\Phi_1 \Rightarrow \Box\Phi_3) \Rightarrow \Box\Phi_1 \Rightarrow (\Box\Box\Phi_1 \Rightarrow \Box\Phi_3)$, which is an instance of (k), and infer $\Box\Phi_1 \Rightarrow \Box\Box\Phi_1 \Rightarrow \Box\Phi_3$ by modus ponens. Now, produce $(\Box\Phi_1 \Rightarrow \Box\Box\Phi_1 \Rightarrow \Box\Phi_3) \Rightarrow (\Box\Phi_1 \Rightarrow \Box\Box\Phi_1) \Rightarrow (\Box\Phi_1 \Rightarrow \Box\Phi_3)$, which is an instance of (s), and infer $(\Box\Phi_1 \Rightarrow \Box\Box\Phi_1) \Rightarrow (\Box\Phi_1 \Rightarrow \Box\Phi_3)$ by modus ponens. But $\Box\Phi_1 \Rightarrow \Box\Box\Phi_1$ is an instance of (4), from which we deduce $\Box\Phi_1 \Rightarrow \Box\Phi_3$, i.e. $\Box\Phi_1 \Rightarrow \Phi_2$, by modus ponens.

If Φ_2 was an instance of (K), (T), or (4), then produce $\Phi_2 \Rightarrow \Box\Phi_1 \Rightarrow \Phi_2$, which is an instance of (k), and apply modus ponens to get a proof of $\Box\Phi_1 \Rightarrow \Phi_2$. □

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

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 $Bxyz = x(yz)$ as $Bxy = S(Kx)y$.) □

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

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

Lemma A.6 *For any formulas* $\Phi_1, \Phi_2, \ldots, \Phi_n$, *and* $1 \leq i \leq n$, *we have* , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_i$.

Proof: Without loss of generality, we can assume $i = 1$: if $i > 1$, we construct a proof of , $\vdash \Phi_i \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_i$, from which we deduce a proof of , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_i$ by Lemma A.5.

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

Assume that $\Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi_1$ is provable from , for all formulas Φ_1, Φ_2, ..., Φ_n; we show that this holds, too, when n is replaced by $n+1$. So, take $n+1$ formulas Φ_1, Φ_2, ..., Φ_{n+1}. Then $\Phi_1 \Rightarrow \Phi_3 \Rightarrow \Phi_4 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi_1$ is provable. Let's name Ψ the formula $\Phi_3 \Rightarrow \Phi_4 \Rightarrow \ldots \Rightarrow \Phi_{n+1} \Rightarrow \Phi_1$, so that $\Phi_1 \Rightarrow \Psi$ is provable.

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

□

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

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

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* , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi_1 \Rightarrow \Psi_2$ *and* , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi_2 \Rightarrow \Psi_3$. *Then* , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi_1 \Rightarrow \Psi_3$.

Proof: Same proof as Lemma A.4, using Lemma A.7 instead of modus ponens. □

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

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

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

$(\Rightarrow R)$ If $\Box$, , $\Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi$ is derived by $(\Rightarrow R)$, then Φ is $\Psi \Rightarrow \Psi'$, and $\Box$, , $\Phi_1, \Phi_2, \ldots, \Phi_n, \Psi \vdash \Psi'$ is derivable. By induction hypothesis, $\Box$, $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Phi_n \Rightarrow \Psi \Rightarrow \Psi'$, i.e. $\Box$, $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Phi_n \Rightarrow \Phi$.

(Cut) If $\Box, \,, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi$ is derived by (Cut), then by induction hypothesis , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi$ and , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi \Rightarrow \Phi$, for some formula Φ. We then conclude by applying Lemma A.7.

($\Box L$) If $\Box, \,, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi$ is derived by ($\Box L$), then we can assume that , $=$, $'$, Φ_0, and that by induction hypothesis , $' \vdash \Phi_0 \Rightarrow \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi$. In particular, , $\vdash \Phi_0 \Rightarrow \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi$. But , $\vdash \Phi_0$ by assumption, so by modus ponens , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Phi$.

($\Box R$) If $\Box, \,, \Phi_1, \Phi_2, \ldots, \Phi_n \vdash \Phi$ is derived by ($\Box R$), then Φ is $\Box\Psi$, and $\Box, \,, \,, ' \vdash \Psi$ is derivable, where , $'$ is the subset $\Phi'_1, \ldots, \Phi'_m$, $m \leq n$, of formulas in $\Phi_1, \ldots, \Phi_n$ that are boxed. The ($\Box R$) rule then infers $\Box, \,, \,, '', \,, ' \vdash \Phi$, where , $''$ is the set $\Phi''_1, \ldots, \Phi''_{n-m}$ of non boxed formulas in $\Phi_1, \ldots, \Phi_n$. By induction hypothesis, , $\vdash \Phi'_1 \Rightarrow \ldots \Rightarrow \Phi'_m \Rightarrow \Psi$ is provable, and we want to show that , $\vdash \Phi_1 \Rightarrow \Phi_2 \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Box\Psi$ is provable, i.e. , $\vdash \Phi''_1 \Rightarrow \ldots \Rightarrow \Phi''_{n-m} \Rightarrow \Phi'_1 \Rightarrow \ldots \Rightarrow \Phi'_m \Rightarrow \Box\Psi$ is provable.

First, we prove by necessitation , $\vdash \Box(\Phi'_1 \Rightarrow \ldots \Rightarrow \Phi'_m \Rightarrow \Psi)$, then we use modus ponens with (K) m times to prove , $\vdash \Box\Phi'_1 \Rightarrow \ldots \Rightarrow \Box\Phi'_m \Rightarrow \Box\Psi$. We now prove that , $\vdash \Phi'_1 \Rightarrow \ldots \Phi'_{i-1} \Rightarrow \Box\Phi'_i \Rightarrow \ldots \Box\Phi'_m \Rightarrow \Box\Psi$ by induction on i. If $i = 0$, this is the result above. So assume that , $\vdash \Phi'_1 \Rightarrow \ldots \Phi'_{i-1} \Rightarrow \Box\Phi'_i \Rightarrow \ldots \Box\Phi'_m \Rightarrow \Box\Psi$ is provable. Then, $\Phi'_i \Rightarrow \Box\Phi'_i$ is provable by (4) because Φ'_i is boxed. By Lemma A.8, we get , $\vdash$ $\Phi'_1 \Rightarrow \ldots \Phi'_{i-1} \Rightarrow \Phi'_i \Rightarrow \Box\Phi'_{i+1} \Rightarrow \ldots \Box\Phi'_m \Rightarrow \Box\Psi$.

So, now in particular we can prove , $\vdash \Phi'_1 \Rightarrow \ldots \Phi'_m \Rightarrow \Box\Psi$, and by Lemma A.5, the desired result , $\vdash \Phi''_1 \Rightarrow \ldots \Rightarrow \Phi''_{n-m} \Rightarrow \Phi'_1 \Rightarrow \ldots \Rightarrow \Phi'_m \Rightarrow \Box\Psi$.

QED.

B Properties of the λ›‹-Calculus

Theorem B.1 *The λ›‹-calculus is confluent.*

Proof: Consider the reduction R defined by all rules but (β). (β) 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 $\xrightarrow{\beta}^*$ and $\xrightarrow{R}^*$ of (β) 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, v, v' such that $u\xrightarrow{\beta}v$ and $u\xrightarrow{R}v'$. The redex r that is contracted to get v is disjoint from the redex r' that 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 v reduces to w, and v' reduces in at most one step to w. So $\xrightarrow{\beta}^*$ and $\xrightarrow{R}^*$ (this is proposition 3.3.6 in [Bar84]). $\Box$

Theorem B.2 *All typed λ›‹-terms are strongly normalizing.*

Proof: We shall use the fact that simply-typed λ-terms are strongly normalizing, and translate λ›‹-terms of the form ›u and ‹u to $(\lambda x \cdot x)u$. Typable λ›‹-terms of some type then translate to typable λ-terms of the same type, where all $\Box$ symbols have been erased.

Let's write $\hat{u}$ 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 λ-redex as the same reduction steps again $\hat{u}_i \to \hat{u}_{i+1}$; and we rewrite ›‹ reduction steps (where ›(‹u) rewrites to u) as two β-reduction steps $(\lambda x \cdot x)(\lambda x \cdot x)\hat{u} \to (\lambda x \cdot x)\hat{u} \to \hat{u}$. 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, too. $\Box$

Definition B.1 *We say that a λ›‹-term is an* elimination *if and only if it is of the form vw or ›v.*

Let u be a λ›‹-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 ›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 vw is the sum of those of v and w 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 $(u_i)_{i\geq 0}$ 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. $\Box$

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

Proof: Define the sequence $(u_i)_{i\geq 0}$ as in Definition B.1, and let $n \geq 1$ be the index at which it stops. Then u_n is not an elimination.

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

If u_{n-1} is an evaluation ›u_n, then u_n cannot be an abstraction by typing, or a quotation by normality. So, in both cases, u_n must be a variable x.

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

Theorem B.5 *Every typable normal λ›‹-term labels the end sequent of some cut-free* **LS4** *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 $\lambda x \cdot v$, 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, using $(\Box R)$.

If u is an elimination, 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 stricly smaller than u, too, so we can apply the induction hypothesis and build $\pi(u)$ as:

$$\begin{array}{c} \pi(d_2(u,y)) \qquad\qquad \pi(v) \\ \vdots \qquad\qquad\qquad \vdots \\ (\Rightarrow L)\ \dfrac{,\ , y : \Phi' \vdash d_2(u,y) : \Phi \qquad , \ \vdash v : \Phi''}{,\ , x : \Phi'' \Rightarrow \Phi' \vdash d_2(u,y)[xv/y] : \Phi'} \end{array}$$

- or $d_1(u) =$ ’x for some variable x. We then define $\pi(u)$ as :

$$\begin{array}{c} \pi(d_2(u,y)) \\ \vdots \\ (\Box L)\ \dfrac{,\ , y : \Phi' \vdash d_2(u,y) : \Phi}{,\ , x : \Box\Phi' \vdash d_2(u,y)[\text{’}x/y] : \Phi} \end{array}$$

□

Theorem B.6 *Every typable* λ’‘*-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 α, β, ..., and consider substitutions σ from type meta-variables to types. For any type Φ, $\Phi\sigma$ denotes the result of applying the substitution σ to Φ. We assume that, if , is a context $x_1 : \Phi_1, \ldots, x_m : \Phi_m$, then , σ denotes the context $x_1 : \Phi_1\sigma, \ldots, x_m : \Phi_m\sigma$. $\sigma\sigma'$ denotes the composition of σ and σ', defined as the substitution such that $\Phi(\sigma\sigma') = (\Phi\sigma)\sigma'$ for any Φ. A substitution or type T is more general than T' if and only if T' can be expressed as $T\sigma''$ for some substitution σ''. We extend the notion to couples of such objects in the obvious way.

We define a type inferencing algorithm $T(, , u)$ that returns a most general couple (σ, Φ) such that , $\sigma \vdash u : \Phi$ is derivable, or fails if there is no couple (σ, Φ) making , $\sigma \vdash u : \Phi\sigma$ 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, and solving equality constraints between types by unification.

To infer the most general substitution and type of u under , , we compute $(\sigma, \Phi) = T(, , u)$ if the computation succeeds, or fail. $T(, , u)$ is defined as usual for all constructions of the λ-calculus; only the ‘ and ’ constructions need some additional definitions:

- (Ax) if x is a variable, and , contains an assumption $x : \Phi$, then $T(, , x) = ([], \Phi)$, where $[]$ is the empty substitution; otherwise, $T(, , x)$ fails;
- $(\Rightarrow E)$ to compute $T(, , uv)$, we compute $(\sigma_1, \Phi_1) = T(, , v)$ (and we fail if this fails), then we compute $(\sigma_2, \Phi_2) = T(, \sigma_1, u)$ (and fail if this fails), then let α be a fresh type variable, and σ_3 be the most general unifier of $\Phi_1\sigma_2 \Rightarrow \alpha$ and Φ_2 (we fail if they are not unifiable), then we return $(\sigma_1\sigma_2\sigma_3, \alpha\sigma_3)$;
- $(\Rightarrow I)$ to compute $T(, , \lambda x \cdot u)$, we compute $(\sigma, \Phi) = T((, , x : \alpha), u)$, where α is a fresh type variable, and fail it it fails; otherwise we return $(\sigma, \alpha\sigma \Rightarrow \Phi)$;

- ($\Box E$) to compute $T(, ,‘u)$, we compute $(\sigma_1, \Phi_1) = T(, , u)$ (and fail if it fails), then we let σ_2 be the most general unifier of Φ_1 and $\Box\alpha$, where α is a fresh type variable (we fail if this fails), and return $(\sigma_1\sigma_2, \alpha\sigma_2)$;

- ($\Box I$) to compute $T(, ,‘u)$ is more complicated. First, notice that a sequent $,' \vdash u' : \Phi'$ is derivable only when, for all free (term) variables x in u', there is an assumption $x : \Psi$ in $,'$ (this is an easy structural induction on the derivation). In particular, in the case of a term $‘u$, $, \sigma \vdash ‘u : \Box\Phi$ can only be derivable if, for every free term variable x_i, $1 \leq i \leq n$, in u, there is an assumption of the form $x_i : \Box\Psi_i$ in $,$, and $x_1 : \Box\Psi_1\sigma, \ldots, x_n : \Box\Psi_n\sigma \vdash u : \Phi$ is derivable.

  Therefore, to compute $T(, ,‘u)$, we first check that all the free variables $x_1, \ldots, x_n$ of u have assumptions $x_i : \Psi_i'$, $1 \leq i \leq n$, in $,$. We then compute the most general simultaneous unifier σ_1 of Ψ_1' with $\Box\alpha_1, \ldots,$ Ψ_n' with $\Box\alpha_n$, where $\alpha_1, \ldots, \alpha_n$ are fresh type variables, 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 $, \vdash u : \Phi$ is derivable, then $, \sigma \vdash u : \Phi\sigma$ is derivable for any substitution σ: 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 Φ and Φ' in linear time, and represent them as triangular forms of size at most the sum of the sizes of Φ and Φ' [MM82]. $\Box$

References

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California 1990. January.

[Bar84] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, 1984.

[BdP92] Gavin Bierman and Valeria de Paiva. Intuitionistic necessity revisited. In *Logic at Work*, Amsterdam, the Netherlands, 1992.

[BdP95] Gavin Bierman and Valeria de Paiva. Intuitionistic necessity revisited. *Journal of Symbolic Logic*, 1995. submitted.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *1988 ACM Conference on Lisp and Functional Programming*, pages 86–95, 1988.

[CH87] Thierry Coquand and Gérard P. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In The Paris Logic Group, editor, *Logic Colloquium '85*, pages 123–146. North-Holland Publishing Company, 1987.

[CR91] William Clinger and Jonathan Rees. The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4(3):1–55, 1991.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. Elsevier Science Publishers b.v., 1990.

[DP95] Rowan Davies and Frank Pfenning. A modal analysis of staged computations. In *Workshop on Types for Program Analysis*, Aarhus, Denmark, May 1995.

[GdQ90] Dov M. Gabbay and Ruy J.G.B. de Queiroz. Extending the Curry-Howard interpretation to linear, relevance and other resource logics. In *Logic Colloquium'90*, Helsinki, July 1990. also as preprint of a full paper to appear in the Journal of Symbol Logic, available at the Department of Computing, Imperial College, 44 pp., or by ftp at the ftp archive at `theory.doc.ic.ac.uk/`.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Company, 1971.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Gor93] Rajeev Goré. Semi-analytic tableaux for propositional modal logics of nonmonotonicity. In David Basin, Reiner Hähnle, Bertram Fronhöfer, Joachim Posegga, and Camilla Schwind, editors, *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 89–100, Marseille, France, April 1993. Technical Report MPI-I-93-213, Max-Planck-Iinstitut für Informatik, Saarbrücken, Germany.

[Gri90] Timothy G. Griffin. A formulas-as-types notion of control. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990.

[Hin69] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transations of the American Mathematical Society*, 146:29–60, 1969.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Kri92] Jean-Louis Krivine. *Lambda-calcul, types et modèles.* Masson, 1992.

[MCAE$^+$62] John Mac Carthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MIT95] MIT Artifical 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.", 1995.

[MM82] A. Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[PW95] Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In *11th Conference on Mathematical Foundations of Programming Semantics*, 1995. Extended Abstract.

[Rei94] Andy Reinhardt. The network with smarts—new agent-based WANs presage the future of connected computing. *BYTE*, October 1994. On Telescript, IBM's planned Intelligent Communications Service, with a sidebar about related Internet developments: Secure-HTTP and Safe-Tcl.