ISSN 2007-9737

A Dual-Context Sequent Calculus for $\mathrm{S4}$ Modal Lambda-Term Synthesis

Favio E. Miranda-Perea[1], Sammantha Omaña Silva[2], Lourdes del Carmen González Huesca[1]

[1] Universidad Nacional Autónoma de México,
Facultad de Ciencias,
Departamento de Matemáticas,
Mexico

[2] Universidad Nacional Autónoma de México,
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas,
Posgrado en Ciencia e Ingeniería de la Computación,
Mexico

{favio, luglzhuesca}@ciencias.unam.mx, samm.omana@gmail.com

Abstract. In type-based program synthesis, the search of inhabitants in typed calculi can be seen as a process where a specification, given by a type A, is considered to be fulfilled if we can construct a λ-term M such that $M : A$, or more precisely if $\Gamma \vdash M : A$ holds, that is, if under some suitable assumptions Γ the term M inhabits the type A. In this paper, we tackle this inhabitation/synthesis problem for the case of modal types in the necessity fragment of the constructive logic $\mathrm{S4}$. Our approach is human-driven in the sense of the usual reasoning procedures of modern theorem provers. To this purpose we employ a so-called dual-context sequent calculus, where the sequents have two contexts, originally proposed to capture the notions of global and local truths without resorting to any formal semantics. The use of dual-contexts allows us to define a sequent calculus which, in comparison to other related systems for the same modal logic, exhibits simpler typing inference rules for the $\Box$ operator. In several cases, the task of searching for a term having subterms with modal types is reduced to the quest for a term containing only subterms typed by non modal propositions.

Keywords. Dual-context sequent calculus, constructive necessity, type inhabitation, modal lambda calculus, program synthesis.

1 Introduction

Modal logic, originated in Mathematics and Philosophy, plays nowadays an important role in Computer Science. The use of modalities is relevant in the theory of programming languages where modal formulas of the form $\Box A$ designate a type of encapsulated values, to be considered *enhanced*, related to *ordinary* values of type A.

For instance in staged computation [8] where $\Box A$ is the type of run-time generated code that computes values of type A. Another important reading of modal types comes out in mobile computation [23, 22] where $\Box A$ is the type of mobile code of type A. Other relevant interpretations of the necessity modality appear in the analysis of information flow either in computer networks [6] or in software security [21].

Coming from practical applications in the mentioned areas of computation, attention is focused on abstracting this kind of behaviors from real scenarios through specifications, which generates an outstanding task: the problem of constructing a program from a given specification.

In this paper we tackle a version of this synthesis problem at the foundational level given

ISSN 2007-9737

by lambda-term type-based synthesis. This is an instance of constructive/deductive synthesis [4] where a type A, coming from the constructive modal logic $\mathrm{S4}$ for necessity in our case, plays the role of the specification that the sought after lambda-term has to meet together with a context Γ to represent a collection of subcomponents, specified again by their types, considered as already synthetized.

This approach corresponds to the type inhabitation problem: *given a context Γ of type declarations for variables and a type A, is it possible to find a term M such that the typing $\Gamma \vdash M : A$ holds?* which in turn corresponds, under the Curry-Howard correspondence, to proof-search: *given a context of assumptions Γ and a formula A, find a derivation of the sequent $\Gamma \vdash A$.*

This problem has been addressed before in the case of modal logic [1, 13, 26] but not from the point of view we take here, which is an interactive human-driven process reminiscent of the ways of modern proof-assistants (like Coq [7]).

Let us show an example of the kind of programs (lambda-terms) we want to synthetize.

> The specification $\Box(A \to B) \to \Box A \to \Box B$ corresponds to a program K witnessing the fact that encapsulated values are well-behaved under function application. In staged computation this means that taking as inputs run-time generated codes of types $A \to B$ and A, program K produces a code of type B:
>
> $$\mathsf{K}\ f\ x =_{def} \mathsf{let\ box}\ f' = f\ \mathsf{in},\ \mathsf{let\ box}\ x' = x\ \mathsf{in},\ \mathsf{box}\ (f' \star x').$$
>
> Here the box constructor signals the encapsulation process: if e denotes an encapsulated value then, e' denotes its ordinary associated value so that the equality $\mathsf{box}\ e' = e$ holds. The $\star$ operator corresponds to function application of ordinary values retrieved from their encapsulated versions.

The above program K corresponds to the characteristic scheme of the modal logic $\mathrm{S4}$, namely $\mathbb{K}$. This example emphasizes the distinction between the two kinds of values[1]: ordinary and enhanced. The idea behind an enhanced value is that it does not depend on any ordinary value. In summary we will consider two kinds of values and two processes which can be applied to any value, namely encapsulation and retrieval. All these behaviors will be enforced by the syntax and the type system.

We start in Section 2 by discussing the dual-context sequent calculus $\mathcal{GS}4$, as a lambda-term type system. Our interactive program synthesis procedure is presented in Section 3, including an example. The soundness of the synthesis process is developed in Section 4, followed by some final remarks in Section 5.

2 A Dual-Context Sequent Calculus for Interactive Program-Synthesis

There are several discussions and applications involving deductive systems for modal logic, see [27, 15] for a deep overview. However, to the best of our knowledge, there is no dedicated sequent calculus presentation of constructive $\mathrm{S4}$ with the intention of human-driven proof-search or program synthesis.

Although, there are works on automated proof-search whose formalisms are therefore not suitable for high-level human-reasoning [1, 26, 13, 15, 18]. Let us review some proposed rules for $\mathrm{S4}$ present in the literature, adapted here for the case of constructive logic:

$$\frac{A, \Box A, \Gamma \vdash B}{\Box A, \Gamma \vdash B}\ (\Box\mathsf{L})\ .$$

Although this left rule is adequate for proof-search, keeping both A and $\Box A$ in the premise somehow entails redundant information and poses loop problems for automated proof-search, solved by means of sophisticated systems like the one

[1] Through the whole paper we allow ourselves to speak of values when referring to any expression which, perhaps after an adequate binding, would yield a well-defined value under any fixed dynamic semantics.

in [26]. In the case of right rules we mention two alternatives:

$$\frac{\Gamma^{\Box} \vdash A}{\Gamma^{\Box} \vdash \Box A}\ (\Box R2), \qquad \frac{\Gamma^{\circ} \vdash A}{\Gamma \vdash \Box A}\ (\Box R3),$$

where $\Gamma^{\Box}$ denotes a context with only boxed formulae and the context Γ° results from Γ by eliminating all the non modal formulae.

Apart from discarding the symmetry between left and right rules, rule $(\Box R2)$ is not suitable for proof-search, since it restricts the shape of the assumptions to be boxed formulas.

In the case of rule $(\Box R3)$, the conclusion sequent has the desired general form for proof-search, but in the generated subgoal $\Gamma^{\circ} \vdash A$, we can lose important information by passing from Γ to Γ°, although this can be alleviated by a clever decision taken by a human-agent, in order to avoid search flaws.

To mitigate the above mentioned issues and tackle the problem of type-based program synthesis we propose a sequent calculus which handles sequents of the form $\Delta \mid \Gamma \vdash M : A$ where M is a lambda-term and Δ and Γ are contexts. This kind of formalism for modal logic has its origins in systems for linear logic [3] and has been introduced in [24] for reconstructing modal logic in the light of Martin-Löf's meaning of logical constants and laws [17]. In this approach, propositions obtain their meaning through judgments without any semantic label (worlds), in particular, modal operators are defined by means of judgments over propositions.

The notion of so-called hypothetical judgments is extended to categorical judgments where a conclusion does not depend on hypotheses about the constructive truth of propositions.

Hence, a distinction of two forms of primitive judgments is essential: 'A *true*' means that we know how to verify A under hypothetical judgments, whereas 'A *valid*' represents the fact that A is a proposition whose truth does not depend on any hypotheses, thus internalizing a categorical judgment as a proposition syntactically represented by the modal formula $\Box A$.

A disengagement similar to the context separation in dual-context systems is present in several works, for instance the systems of Fitting [10]; or the work of Avron et al. [2].

We move now to the technical definitions. The types are generated by the following grammar where $\mathcal{B}$ denotes a collection of primitive basic types:

$$A, B ::= \mathcal{B} \mid A \to B \mid A \wedge B \mid A \vee B \mid \Box A.$$

Unlike other presentations, we include disjunction and conjunction. Also note that neither negation nor $\bot$ are present. Thus, we are dealing with minimal logic. Variable declaration contexts are defined by means of so-called *snoc* lists:

$$\Gamma ::= \cdot \mid \Gamma, x : A,$$

these are finite lists built from the empty list, denoted here by $\cdot$, and a binary constructor that generates a new list from a given one by adding a fresh variable x of type A to its right-end. The concatenation operation is inductively defined as expected and denoted by $\Gamma_1; \Gamma_2$.

In the below inference rules the idea behind the context separation is that hypotheses in Δ are enhanced (modal), whereas those in Γ are ordinary (intuitionistic). Nevertheless, this idea does not represent a syntactic restriction, for we can have arbitrary types, modal or intuitionistic, in both contexts. This is an important difference with other modal dual-context systems like those of [16].

Moreover, the context separation is strict, in particular it is forbidden to declare the same variable in both contexts. This is an important difference with [24] where there is only one context with two zones, a choice that requires the use of explicit labels *valid, true* in the context formulae.

Since their introduction, dual-context modal logics and their type systems have been presented in the sequent-style of natural deduction [16, 8, 22, 9].

Such formalisms are not suitable for backward proof-search, reason why we present a sequent calculus $\mathcal{GS}4$, adequate for our purposes. This system is inductively defined by the following inference rules, the corresponding lambda-terms encoding proofs are the target of the synthesis process discussed in detail in the next section:

ISSN 2007-9737

— Initial rules: we have two rules that allow to conclude a hypothesis according to the context it belongs:

$$\frac{}{\Delta \mid \Gamma, x : A; \Gamma' \vdash x : A}\ (\textsc{Thyp})\ ,$$

$$\frac{}{\Delta, x : A; \Delta' \mid \Gamma \vdash x : A}\ (\textsc{Vhyp})\ .$$

— Right rules:

$$\frac{\Delta \mid \Gamma \vdash M : A \qquad \Delta \mid \Gamma \vdash N : B}{\Delta \mid \Gamma \vdash \langle M, N\rangle : A \wedge B}\ (\wedge\,\mathsf{R})\ ,$$

$$\frac{\Delta \mid \Gamma \vdash M : A}{\Delta \mid \Gamma \vdash \mathsf{inl}\, M : A \vee B}\ (\vee\,\mathsf{R})\ ,$$

$$\frac{\Delta \mid \Gamma \vdash M : B}{\Delta \mid \Gamma \vdash \mathsf{inr}\, M : A \vee B}\ (\vee\,\mathsf{R})\ ,$$

$$\frac{\Delta \mid \Gamma, x : A \vdash N : B}{\Delta \mid \Gamma \vdash \lambda x.N : A \to B}\ (\to\mathsf{R})\ ,$$

$$\frac{\Delta \mid \cdot \vdash M : A}{\Delta \mid \Gamma \vdash \mathsf{box}\, M : \Box A}\ (\Box\,\mathsf{R})\ .$$

The rules for propositional connectives are standard. In the case of a modal formula $\Box A$ the right rule corresponds to the so-called necessitation rule and allows us to introduce the box operator on the right hand side of the turnstile, only in the absence of ordinary assumptions.

It is important to remark that this right rule, as rule $(\Box R3)$, also suffers from loss of information in its backward reading.

However, such issue can be avoided in some cases by transferring to Δ some or all the boxed assumptions in Γ, which is not possible with $(\Box R3)$.

The left rules come in two versions, one for each context.

— Left rules for the ordinary context:

$$\frac{\Delta \mid \Gamma, x : A, y : B; \Gamma' \vdash M : C}{\Delta \mid \Gamma, z : A \wedge B; \Gamma' \vdash \mathsf{letpair}(z, x.y.M) : C}\ (\wedge\mathsf{L})\ ,$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma, x : A; \Gamma' \vdash M : C \\ \Delta \mid \Gamma, y : B; \Gamma' \vdash N : C\end{array}}{\Delta \mid \Gamma, z : A \vee B; \Gamma' \vdash \mathsf{case}(z, x.M, y.N) : C}\ (\vee\,\mathsf{L})\ ,$$

$$\frac{\Delta \mid \Gamma, x : A \to B; \Gamma' \vdash M : A}{\Delta \mid \Gamma, x : A \to B; \Gamma' \vdash xM : B}\ (\to\mathsf{L})\ ,$$

$$\frac{\Delta, x : A \mid \Gamma; \Gamma' \vdash M : B}{\Delta \mid \Gamma, y : \Box A; \Gamma' \vdash \mathsf{letbox}(y, x.M) : B}\ (\Box\,\mathsf{L})\ .$$

For disjunction and conjunction, the rules are standard. The left rule $(\Box L)$ represents a type transference principle between contexts: in the proof-search process we can move an encapsulated type in the ordinary context to the enhanced context by unboxing it.

The rule $(\to$ L) is not usual, for instead of decomposing the implicative hypothesis, in order to prove/use its components, like the regular left rule for implication, it only uses it to derive its consequent, once its antecedent has been derived.

This rule captures the local reasoning with implication common in informal proofs, instead the usual left rule for implication models an on the-fly prove/use of a lemma. This feature rarely figures in actual paper-and-pencil proofs and thus makes the original rule clumsy for proof-search purposes.

To the best of our knowledge this left rule, which is inspired by the `apply` tactic of the Coq proof-assistant, has been considered only by us [20, 19], though it is also related to the rule $(\to L)^\circ$ of Schroeder-Heister [28].

Let us also note that, under the presence of the cut or substitution rule, the rule $(\to$L) is equivalent to the ordinary left rule for implication. The details of this claim are omitted due to lack of space.

— Left rules for the enhanced context:

$$\frac{\Delta, x : A, y : B; \Delta' \mid \Gamma \vdash M : C}{\Delta, z : A \wedge B; \Delta' \mid \Gamma \vdash \mathsf{eletpair}(z, x.y.M) : C} \ (\wedge\mathsf{LE}),$$

$$\frac{\begin{array}{c}\Delta; \Delta' \mid \Gamma, x : A \vdash M : C \\ \Delta; \Delta' \mid \Gamma, y : B \vdash N : C\end{array}}{\Delta, z : A \vee B; \Delta' \mid \Gamma \vdash \mathsf{ecase}(z, x.M, y.N) : C} \ (\vee\mathsf{LE}),$$

$$\frac{\Delta, x : A \to B; \Delta' \mid \Gamma \vdash M : A}{\Delta, x : A \to B; \Delta' \mid \Gamma \vdash x \star M : B} \ (\to\mathsf{LE}),$$

$$\frac{\Delta, x : A; \Delta' \mid \Gamma \vdash M : B}{\Delta, y : \Box A; \Delta' \mid \Gamma \vdash \mathsf{eletbox}(y, x.M) : B} \ (\Box\,\mathsf{LE}).$$

The rule for conjunction is again standard, whereas for implication the rule is analogous to the version for ordinary contexts. In the case of an enhanced disjunctive hypothesis the case analysis on $z : A \vee B$ is performed only by ordinary hypotheses $x : A$ and $y : B$, otherwise the rule would be unsound [2]. Finally, the rule $\Box LE$, introduced by us in [19], reduces the synthesis of a program involving an enhanced and encapsulated component $\Box A$ to the search of a program involving only the non-encapsulated enhanced component A.

— Substitution or cut rules: these are essential for human-driven proof-search:

$$\frac{\Delta \mid \Gamma \vdash M : A \qquad \Delta \mid \Gamma, x : A \vdash N : B}{\Delta \mid \Gamma \vdash \mathsf{let}(M, x.N) : B} \ (\textsc{Subst}),$$

$$\frac{\Delta \mid \cdot \vdash M : A \qquad \Delta, x : A \mid \Gamma \vdash N : B}{\Delta \mid \Gamma \vdash \mathsf{elet}(M, x.N) : B} \ (\textsc{SubstE}).$$

The enhanced substitution rule (SUBSTE) is derivable from (SUBST), but we keep both rules for the sake of symmetry. Moreover, it is relevant to mention that this ordinary rule is not admissible. This is not important for our purposes since the reasoning pattern provided by the cut rule is essential for human-driven proof search. A further discussion on this matter has to be presented elsewhere, due to lack of space.

With respect to the left rules for necessity we consider important to emphasize that, since they permit to encapsulate/retrieve values at the hypotheses level, the synthesis process involving an enhanced value can be reduced to one that requires only an ordinary value. This transference process (see [11, Section 4.2]) allows us to reason in a more intuitive and straighforward way, as shown in the example of Section 3.

To finish the section is important to review the more usual elimination rule for $\Box$ in dual-context systems presented for instance in [25, 8, 16, 9]. This typing rule for a letbox operator is:

$$\frac{\Delta \mid \Gamma \vdash M : \Box A \qquad \Delta, x : A \mid \Gamma \vdash N : B}{\Delta \mid \Gamma \vdash \mathsf{letb}\, x = M \,\mathsf{in}\, N : B} \ (\Box\ \mathsf{E}).$$

We can observe that such rule entails a specific substitution (cut) process that provides the primitive way of using an ordinary value (the assumption $x : A$) retrieved from an enhanced value (the term $M : \Box A$), at the price of requiring an explicit derivation of M. Of course this is characteristic of (generalized) elimination rules in sequent-style natural deduction and can be simulated in $\mathcal{GS}4$ by means of the $(\Box L)$ and (SUBST) rules. The definition of letb, in concrete syntax, witnessing this simulation is: $\mathsf{letb}\, x = M \,\mathsf{in}\, N =_{def} \mathsf{let}\, y = M \,\mathsf{in}\, \mathsf{letbox}\, x = y \,\mathsf{in}\, N$.

It is easy to see that in a dual-context natural deduction system, with $(\Box E)$ as a primitive rule, our rules $(\Box L)$ and $(\Box LE)$ can be simulated as well. Thus, both systems turn out to be equivalent (more details are provided in [19]). Moreover, in [11] we prove that the system of dual-context natural deduction is equivalent to an S4-axiomatic system. Then we can conclude that the present system captures exactly the necessity fragment of the constructive logic S4.

3 Interactive Program Synthesis

The lambda-terms that appear in the sequent calculus typing rules of Section 2, formalize the kind of programs target of the synthesis process. They include modal constructors in the lines of some related languages [25, 8, 16, 9] as well as distinct variable binding operators (let or case expressions). Let us collect them precisely.

[2] For instance, it would allow to derive $\Box(A \vee B) \to \Box A \vee \Box B$, which is invalid in all known semantics of S4.

ISSN 2007-9737

Definition 3.1. *A pseudoterm is an expression generated by the following grammar:*

$$\begin{array}{rcl} M & ::= & x \mid X \mid R \mid O \mid E \mid P \\ R & ::= & \lambda x.M \mid \langle M, M\rangle \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M \mid \mathsf{box}\, M \\ O & ::= & M\, M \mid \mathsf{letpair}(M, x.y.M) \mid \\ & & \mathsf{case}(M, x.M, y.M) \mid \mathsf{letbox}(M, y.M) \\ E & ::= & M \star M \mid \mathsf{eletpair}(M, x.y.M) \mid \\ & & \mathsf{ecase}(M, x.M, y.M) \mid \mathsf{eletbox}(M, y.M) \\ P & ::= & \mathsf{let}(M, x.M) \mid \mathsf{elet}(M, x.M) \end{array}$$

The metavariable M denotes pseudoterms which are classified as right pseudoterms, those generated by the metavariable R; ordinary pseudoterms, generated by O and enhanced pseudoterms, generated by E. A left pseudoterm is either an ordinary or an enhanced pseudoterm. Finally, those generated by P are called strong let-expressions.

As our matter of interest is to synthetize lambda-terms we define pseudoterms, which are expressions corresponding to programs with unknown templates to be filled during the synthesis process. A basic and least informative template is represented by a category of metavariables or search-variables, disjoint from the usual term variables, denoted by a capital X. A pseudoterm M is called a *term* or a *program* if M does not contain search variables.

Apart from the usual lambda term constructors for functions, sums and products, we use twin program constructors in order to make explicit the manipulation of ordinary or enhanced values, the only difference being a prefix e indicating the need for an enhanced input (we use an infix application operator $\star$ in the case of an enhanced function). There is no need to have twin constructors for right pseudoterms, due to the fact that an enhanced value can be constructed directly by the box operator.

Let us sketch next our program synthesis technique: given two contexts of type declarations for variables, Δ and Γ (for enhanced and ordinary assumptions respectively) and a type specification A, the goal is to construct a program M such that the typing $\Delta \mid \Gamma \vdash M : A$ holds. The program M is currently unknown and it is represented with a search-variable X. The task is to find a value for X such that $\Delta \mid \Gamma \vdash X : A$ holds.

Analyzing the specific form either of A or of some assumption type; and applying a backward reading of the typing rules, some restrictions on the form of M are generated in order to give a solution for X. These conditions are given by pseudoterm equations which, if solvable, will allow to construct the desired program.

For instance, the search for an X such that $\cdot \mid x : A \vdash X : A \vee B$ holds, is reduced to the search for a Y such that $\cdot \mid x : A \vdash Y : A$ holds. The search-variables X and Y are related by the equation $X \approx \mathsf{inl}\, Y$. The last goal is directly solved by the equation $Y \approx x$. By solving these two equations, we obtain that $M =_{def} \mathsf{inl}\, x$ verifies $\cdot \mid x : A \vdash M : A \vee B$.

However, let us observe that sequents involving search-variables are not derivable, for, according to Section 2, there is no typing rule for this kind of variables. Such underivable sequents represent synthesis problems, which are program search tasks formalized by the following notion of pseudosequent.

Definition 3.2. *A pseudosequent or search-sequent, $\mathcal{P}$, is a $4-$tuple of the form $\Delta \mid \Gamma \vdash_? X : A$ where X is a search-variable.*

For the synthesis process we will need to handle finite sequences of pseudosequents defined as follows:

Definition 3.3. *The set of finite sequences of pseudosequents is recursively defined with the grammar:*

$$\mathcal{S} ::= \bullet \mid \mathcal{P}\,,\, \mathcal{S}$$

where $\bullet$ denotes the empty sequence. For clarity, a singleton sequence is identified with its unique element. As for contexts, the semicolon operator ; is used for concatenation of pseudosequents sequences.

The restrictions generated during the program search will be captured by constraint sets defined as follows:

Definition 3.4. *A constraint is an equation of the form $X \approx e$ where X is a search-variable and e is a pseudoterm. A set of equations:*

$$\mathcal{R} = \{X_1 \approx e_1, X_2 \approx e_2, \ldots, X_k \approx e_k\},$$

where all X_i are different, is called a constraint set. According to the category of the pseudoterm e, a constraint can be right, ordinary, enhanced, left or strong.

Next we define what is a solution of a constraint set.

Definition 3.5. *Given a constraint set $\mathcal{R}$, a pseudoterm M is solution of the constraint $X \approx e$ in $\mathcal{R}$, if M is a term and there is a substitution*[3] *of search-variables by terms, say $\sigma = [X_1, \ldots, X_n/M_1, \ldots, M_n]$, such that $M \equiv e\sigma$ (i.e M is syntactically identical to $e\sigma$ up-to α-equivalence). In such case the solution is written as $M = Sol_{\mathcal{R}}(X \approx e)$.*

Sequences of pseudosequents interact with constraint sets by means of goals, defined as follows.

Definition 3.6. *A goal is a pair $\mathcal{S} \parallel \mathcal{R}$ consisting of a sequence of pseudosequents $\mathcal{S}$ and a constraint set $\mathcal{R}$. The set of goals is denoted by* Goal.

The program synthesis process is defined next by means of a transition system where the goals play the role of states and the transitions, which are called tactics, transform a goal into another goal according to the backward reading of the typing rules. We consider this formal definition and handling of backward lambda-term synthesis as the second main contribution of this paper.

Definition 3.7. *The transition system is defined as follows:*

- *A state is a goal $\mathcal{S} \parallel \mathcal{R}$.*
- *An initial state is a goal of the form $\Delta \mid \Gamma \vdash_? X : A \parallel \varnothing$, that is, a goal composed of a unique pseudosequent and the empty constraint set.*
- *A terminal state is a goal of the form $\bullet \parallel \mathcal{R}$, that is, a goal composed of the empty sequence of pseudosequents and an arbitrary constraint set.*
- *The transition relation $\rhd \subseteq$ Goal $\times$ Goal is inductively defined by the axioms and inference rule below, where a transition $\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd \mathcal{S}_2 \parallel \mathcal{R}_2$ can be read as* to solve the current goal $\mathcal{S}_1 \parallel \mathcal{R}_1$ it suffices to solve the subgoal $\mathcal{S}_2 \parallel \mathcal{R}_2$.

In each basic transition, the search-sequent $\Delta \mid \Gamma \vdash_? X : A$ dictates the action according to the backward reading of a typing rule, updating the constraint set accordingly.

In the following, we define the transition system axioms according to four synthesis process. In each case we assume that the search-variables introduced in the pseudosequents of the reduct are fresh, that is, do not occur in the redex.

3.1 Direct Synthesis

Direct synthesis triggers the synthesis by directly analyzing the shape of the typing specification producing a right constraint:

$$\begin{array}{l}
\texttt{intro}\ x: \\
\Delta \mid \Gamma \vdash_? X : A \to B \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma, x : A \vdash_? X_1 : B \parallel \mathcal{R}, X \approx \lambda x.X_1, \\
\texttt{split}: \\
\Delta \mid \Gamma \vdash_? X : A \wedge B \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma \vdash_? X_1 : A\ ; \\
\quad \Delta \mid \Gamma \vdash_? X_2 : B \parallel \mathcal{R}, X \approx \langle X_1, X_2 \rangle, \\
\texttt{left}: \\
\Delta \mid \Gamma \vdash_? X : A \vee B \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma \vdash_? X_1 : A \parallel \mathcal{R}, X \approx \mathsf{inl}\, X_1, \\
\texttt{right}: \\
\Delta \mid \Gamma \vdash_? X : A \vee B \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma \vdash_? X_1 : B \parallel \mathcal{R}, X \approx \mathsf{inr}\, X_1, \\
\texttt{unbox}: \\
\Delta \mid \Gamma \vdash_? X : \Box A \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \cdot \vdash_? X_1 : A \parallel \mathcal{R}, X \approx \mathsf{box}\, X_1,
\end{array}$$

[3] Substitution in the usual sense.

ISSN 2007-9737

3.2 Indirect Synthesis

Indirect synthesis focuses on a type in any of the contexts, generating a left constraint:

$$\begin{array}{l}
\texttt{apply } x: \\
\Delta \mid \Gamma, x : A \to B; \Gamma' \vdash_? X : B \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma, x : A \to B; \Gamma' \vdash_? X_1 : A \parallel \mathcal{R}, X \approx x X_1, \\
\texttt{apply } x: \\
\Delta, x : A \to B; \Delta' \mid \Gamma \vdash_? X : B \parallel \mathcal{R} \rhd \\
\quad \Delta, x : A \to B; \Delta' \mid \Gamma \vdash_? X_1 : A \parallel \mathcal{R}, X \approx x \star X_1, \\
\texttt{destruct } z: \\
\Delta \mid \Gamma, z : A \wedge B; \Gamma' \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma, x : A, y : B; \Gamma' \vdash_? X_1 : C \parallel \mathcal{R}, X \approx \mathsf{letpair}(z, x.y.X_1), \\
\texttt{destruct } z: \\
\Delta, z : A \wedge B; \Delta' \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta, x : A, y : B; \Delta' \mid \Gamma \vdash_? X_1 : C \parallel \mathcal{R}, X \approx \mathsf{eletpair}(z, x.y.X_1), \\
\texttt{destruct } z: \\
\Delta \mid \Gamma, z : A \vee B; \Gamma' \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma, x : A; \Gamma' \vdash_? X_1 : C ; \\
\quad \Delta \mid \Gamma, y : B; \Gamma' \vdash_? X_2 : C \parallel \mathcal{R}, X \approx \mathsf{case}(z, x.X_1, y.X_2), \\
\texttt{destruct } z: \\
\Delta, z : A \vee B; \Delta' \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta; \Delta' \mid \Gamma, x : A \vdash_? X_1 : C ; \\
\quad \Delta; \Delta' \mid \Gamma, y : B \vdash_? X_2 : C \parallel \mathcal{R}, X \approx \mathsf{ecase}(z, x.X_1, y.X_2),
\end{array}$$

$$\begin{array}{l}
\texttt{retrieve } x: \\
\Delta \mid \Gamma, x : \Box A; \Gamma' \vdash_? X : B \parallel \mathcal{R} \rhd \\
\quad \Delta, y : A \mid \Gamma; \Gamma' \vdash_? X_1 : B \parallel \mathcal{R}, X \approx \mathsf{letbox}(x, y.X_1), \\
\texttt{retrieve } x: \\
\Delta, x : \Box A; \Delta' \mid \Gamma \vdash_? X : B \parallel \mathcal{R} \rhd \\
\quad \Delta, y : A; \Delta' \mid \Gamma \vdash_? X_1 : B \parallel \mathcal{R}, X \approx \mathsf{eletbox}(x, y.X_1).
\end{array}$$

In this case, distinct tactics have the same name for they share the same functionality: the `apply` tactics correspond to the application of a functional hypothesis; the `destruct` tactics trigger the destruction of a specific hypothesis, replacing it by simpler hypotheses in the subgoals. Finally, the manipulation of modal hypotheses is managed by the `retrieve` tactics.

3.3 Strong Synthesis

Strong synthesis requires to guess the type corresponding to the first premise of a substitution rule and generates a strong constraint. This calls for an explicit interaction [4] with the human agent, which is why we speak of a strong synthesis:

[4] Nevertheless, the reader can note that the indirect synthesis process also requires interactivity in order to choose an adequate hypothesis and match a particular tactic.

$$\begin{array}{l}
\texttt{assert A}: \\
\Delta \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma \vdash_? X_1 : A ; \\
\quad \Delta \mid \Gamma, x : A \vdash_? X_2 : C \parallel \mathcal{R}, X \approx \mathsf{let}(X_1, x.X_2), \\
\texttt{enough A}: \\
\Delta \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \Gamma, x : A \vdash_? X_1 : C; \\
\quad \Delta \mid \Gamma \vdash_? X_2 : A \parallel \mathcal{R}, X \approx \mathsf{let}(X_2, x.X_1), \\
\texttt{eassert A}: \\
\Delta \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta \mid \cdot \vdash_? X_1 : A ; \\
\quad \Delta, x : A \mid \Gamma \vdash_? X_2 : C \parallel \mathcal{R}, X \approx \mathsf{elet}(X_1, x.X_2), \\
\texttt{eenough A}: \\
\Delta \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \\
\quad \Delta, x : A \mid \Gamma \vdash_? X_1 : C ; \\
\quad \Delta \mid \cdot \vdash_? X_2 : A \parallel \mathcal{R}, X \approx \mathsf{elet}(X_2, x.X_1).
\end{array}$$

Let us observe that each substitution rule has two corresponding tactics, namely `assert` and `enough`, the difference being only operational: either we first synthetize the auxiliar component A and then invoke it or viceversa.

3.4 Immediate Synthesis

Immediate synthesis generates a synthesis problem which is trivially solved by an initial inference rule. The constraints generated here are necessarily of the form $X \approx x$.

$$\begin{array}{l}
\texttt{assumption}: \\
\Delta \mid \Gamma, x : A; \Gamma' \vdash_? X : A \parallel \mathcal{R} \rhd \bullet \parallel \mathcal{R}, X \approx x, \\
\texttt{eassumption}: \\
\Delta, x : A; \Delta' \mid \Gamma \vdash_? X : A \parallel \mathcal{R} \rhd \bullet \parallel \mathcal{R}, X \approx x.
\end{array}$$

The names of some of the above tactics, though not the ones involving modal types, coincide with the names of analogous tactics implemented in the Coq proof assistant.

3.5 Inference Rule for Transition

All the above tactics constitute the basic axioms of the transition system. Next we give the sole inference rule for transitions.

Sequencing:

Several basic transitions cause the sequence of search-sequents in a goal to grow. In such cases we need to choose a specific search-sequent within the current goal to perform the next transition. The sequencing rule (seq) determines the choice order, namely from the first (left most) search-sequent:

$$\frac{\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd \mathcal{S}_2 \parallel \mathcal{R}_2}{\mathcal{S}_1; \mathcal{S} \parallel \mathcal{R}_1 \rhd \mathcal{S}_2; \mathcal{S} \parallel \mathcal{R}_2}(\mathsf{seq}).$$

We show an example of how the transition system of tactics performs the synthesis process. Let us recall that the solution of a constraint is not unique and depends on the solutions of the later constraints found in the synthesis process. The application of any tactic guarantees that each search-variable appears in a pseudoterm in the constraint set.

Example 3.1. *A program t corresponding to the specification given by the modal scheme $\mathbb{K}$ is synthetized as follows.*

$\cdot \mid \cdot \vdash_? X : \Box(A \to B) \to \Box A \to \Box B \parallel \emptyset,$
$\rhd$ `intro` x
$\cdot \mid x : \Box(A \to B) \vdash_? X_1 : \Box A \to \Box B \parallel X \approx \lambda x.X_1,$
$\rhd$ `intro` y
$\cdot \mid x : \Box(A \to B), y : \Box A \vdash_? X_2 : \Box B \parallel \mathcal{R}, X_1 \approx \lambda y.X_2,$
$\rhd$ `retrieve` x
$x_1 : A \to B \mid y : \Box A \vdash_? X_3 : \Box B \parallel \mathcal{R}', X_2 \approx \mathsf{letbox}(x, x_1.X_3),$
$\rhd$ `retrieve` y
$x_1 : A \to B, y_1 : A \mid \cdot \vdash_? X_4 : \Box B \parallel \mathcal{R}'', X_3 \approx \mathsf{letbox}(y, y_1.X_4),$
$\rhd$ `unbox`
$x_1 : A \to B, y_1 : A \mid \cdot \vdash_? X_5 : B \parallel \mathcal{R}''', X_4 \approx \mathsf{box}\, X_5,$
$\rhd$ `apply` x_1
$x_1 : A \to B, y_1 : A \mid \cdot \vdash_? X_6 : A \parallel \mathcal{R}^{iv}, X_5 \approx x_1 \star X_6,$
$\rhd$ `eassumption`,
$\bullet \parallel \mathcal{R}''', X_4 \approx \mathsf{box}\, X_5, X_5 \approx x_1 \star X_6, X_6 \approx y_1,$

where

$$\begin{array}{rcl} \mathcal{R} & = & X \approx \lambda x.X_1, \\ \mathcal{R}' & = & X \approx \lambda x.X_1,\ X_1 \approx \lambda y.X_2, \\ \mathcal{R}'' & = & \mathcal{R}', X_2 \approx \mathsf{letbox}(x, x_1.X_3), \\ \mathcal{R}''' & = & \mathcal{R}'', X_3 \approx \mathsf{letbox}(y, y_1.X_4), \\ \mathcal{R}^{iv} & = & \mathcal{R}''', X_4 \approx \mathsf{box}\, X_5, \end{array}$$

and the solution for X is:
$t =_{def} \lambda x.\lambda y.\, \mathsf{letbox}(x, x.\, \mathsf{letbox}(y, y.\, \mathsf{box}(x_1 \star y_1)))$.

Example 3.2. *A program M corresponding to the specification given by type $\Box(A \wedge B) \to (\Box A \wedge \Box B)$ is synthetized as follows.*

$\cdot \mid \cdot \vdash_? X : \Box(A \wedge B) \to (\Box A \wedge \Box B) \parallel \emptyset,$
$\rhd$ `intro` x
$\cdot \mid x : \Box(A \wedge B) \vdash_? X_1 : \Box A \wedge \Box B \parallel X \approx \lambda x.X_1,$
$\rhd$ `retrieve` x
$x_1 : A \wedge B \mid \cdot \vdash_? X_2 : \Box A \wedge \Box B \parallel \mathcal{R}, X_1 \approx \mathsf{letbox}(x, x.X_2),$
$\rhd$ `destruct` x_1
$y_1 : A, y_2 : B \mid \cdot \vdash_? X_3 : \Box A \wedge \Box B \parallel \mathcal{R}', X_2 \approx \mathsf{eletpair}(x, y_1.y_2.X_3),$
$\rhd$ `split`
$y_1 : A, y_2 : B \mid \cdot \vdash_? X_4 : \Box A$;
$y_1 : A, y_2 : B \mid \cdot \vdash_? X_5 : \Box B \parallel \mathcal{R}'', X_3 \approx \langle X_4, X_5 \rangle,$
$\rhd$ `unbox`,
$y_1 : A, y_2 : B \mid \cdot \vdash_? X_6 : A$;
$y_1 : A, y_2 : B \mid \cdot \vdash_? X_5 : \Box B \parallel \mathcal{R}'', X_3 \approx \langle X_4, X_5 \rangle, X_4 \approx \mathsf{box}\, X_6,$
$\rhd$ `eassumption`
$\bullet$; $y_1 : A, y_2 : B \mid \cdot \vdash_? X_5 : \Box B \parallel \mathcal{R}''', X_6 \approx y_1,$
$\rhd$ `unbox`
$y_1 : A, y_2 : B \mid \cdot \vdash_? X_7 : B \parallel \mathcal{R}''', X_6 \approx y_1, X_5 \approx \mathsf{box}\, X_7,$
$\rhd$ `eassumption`
$\bullet \parallel \mathcal{R}''', X_6 \approx y_1, X_5 \approx \mathsf{box}\, X_7, X_7 \approx y_2,$

where

$$\begin{array}{rcl} \mathcal{R} & = & X \approx \lambda x.X_1, \\ \mathcal{R}' & = & \mathcal{R}, X_1 \approx \mathsf{letbox}(x, x_1.X_2), \end{array}$$

$$\begin{array}{rcl} \mathcal{R}'' & = & \mathcal{R}', X_2 \approx \mathsf{eletpair}(x_1, y_1.y_2.X_3), \\ \mathcal{R}''' & = & \mathcal{R}'', X_3 \approx \langle X_4, X_5 \rangle, X_4 \approx \mathsf{box}\, X_6, \end{array}$$

and the solution for X is:
$M =_{def} \lambda x.\, \mathsf{letbox}(x, x_1.\, \mathsf{eletpair}(x_1, y_1.y_2.\langle \mathsf{box}\, y_1, \mathsf{box}\, y_2 \rangle))$

Example 3.3. *A program M meeting the specification given by $\Box(\Box A \vee \Box B) \to \Box(A \vee B)$ is synthetized as follows:*

$\cdot \mid \cdot \vdash_? X : \Box(\Box A \vee \Box B) \to \Box(A \vee B) \parallel \emptyset,$
$\rhd$ `intro` x
$\cdot \mid x : \Box(\Box A \vee \Box B) \vdash_? X_1 : \Box(A \vee B) \parallel X \approx \lambda x.X_1,$
$\rhd$ `retrieve` x
$x_1 : \Box A \vee \Box B \mid \cdot \vdash_? X_2 : \Box(A \vee B) \parallel \mathcal{R}, X_1 \approx \mathsf{letbox}(x, x_1.X_2),$
$\rhd$ `unbox`
$x_1 : \Box A \vee \Box B \mid \cdot \vdash_? X_3 : A \vee B \parallel \mathcal{R}', X_2 \approx \mathsf{box}\, X_3,$
$\rhd$ `destruct` x_1,
$\cdot \mid x_2 : \Box A \vdash_? X_4 : A \vee B$;
$\cdot \mid x_3 : \Box B \vdash_? X_5 : A \vee B \parallel \mathcal{R}'', X_3 \approx \mathsf{ecase}(x_1, x_2.X_4, x_3.X_5),$
$\rhd$ `left`
$\cdot \mid x_2 : \Box A \vdash_? X_6 : A$;
$\cdot \mid x_3 : \Box B \vdash_? X_5 : A \vee B \parallel \mathcal{R}''', X_4 \approx \mathsf{inl}\, X_6,$
$\rhd$ `retrieve` x_2
$x_4 : A \mid \cdot \vdash_? X_7 : A$;
$\cdot \mid x_3 : \Box B \vdash_? X_5 : A \vee B \parallel \mathcal{R}^{iv}, X_6 \approx \mathsf{letbox}(x_2, x_4.X_7),$
$\rhd$ `eassumption`
$\bullet$; $\cdot \mid x_3 : \Box B \vdash_? X_5 : A \vee B \parallel \mathcal{R}^{v}, X_7 \approx x_4,$
$\rhd$ `right`
$\cdot \mid x_3 : \Box B \vdash_? X_8 : B \parallel \mathcal{R}^{v}, X_7 \approx x_4, X_5 \approx \mathsf{inr}\, X_8$
$\rhd$ `retrieve` x_3
$x_5 : B \mid \cdot \vdash_? X_9 : B \parallel \mathcal{R}^{vi}, X_8 \approx \mathsf{letbox}(x_3, x_5.X_9),$
$\rhd$ `eassumption`
$\bullet \parallel \mathcal{R}^{vi}, X_8 \approx \mathsf{letbox}(x_3, x_5.X_9), X_9 \approx x_5,$

ISSN 2007-9737

where

$$\begin{array}{rcl}\mathcal{R} & = & X \approx \lambda x.X_1,\\ \mathcal{R}' & = & \mathcal{R},\ X_1 \approx \mathsf{letbox}(x, x_1.X_2),\\ \mathcal{R}'' & = & \mathcal{R}',\ X_2 \approx \mathsf{box}\, X_3,\\ \mathcal{R}''' & = & \mathcal{R}'',\ X_3 \approx \mathsf{ecase}(x_1, x_2.X_4, x_3.X_5),\\ \mathcal{R}^{iv} & = & \mathcal{R}''',\ X_4 \approx \mathsf{inl}\, X_6,\\ \mathcal{R}^{v} & = & \mathcal{R}^{iv},\ X_6 \approx \mathsf{letbox}(x_2, x_4.X_7),\\ \mathcal{R}^{vi} & = & \mathcal{R}^{v}, X_7 \approx x_4,\ X_5 \approx \mathsf{inr}\, X_8,\end{array}$$

and the solution for X is:

$$M =_{def} \lambda x.\, \mathsf{letbox}(x, x_1.\, \mathsf{box}(\mathsf{ecase}(x_1, x_2.\, \mathsf{inl}\ \mathsf{letbox}(x_2, x_4.x_4), x_3.\, \mathsf{inr}\, \mathsf{letbox}(x_3, x_5.x_5))))$$

Let us observe that, since the `unbox` and `retrieve` tactics replace a modal with a pure propositional assumption, as announced, we are replacing a modal reasoning with a propositional inference.

4 Soundness of the Synthesis Process

In this section we prove the soundness of the synthesis process. Given an initial goal $\Delta \mid \Gamma \vdash_? X : A \parallel \varnothing$ we want to guarantee that: if the transition relation succeeds, that is, if applying the transition rules a finite number of times from this goal, we arrive to a final goal $\bullet \parallel \mathcal{R}$, then there is a program M, constructed by solving the contraints in $\mathcal{R}$, such that $\Delta \mid \Gamma \vdash M : A$ holds. Let us start by stating a convenient definition for the transitive closure of the transition relation.

Definition 4.1. *The transitive closure of the relation $\rhd$, denoted $\rhd^+$, is inductively defined by the following rules:*

$$\frac{\mathcal{S} \parallel \mathcal{R} \rhd \mathcal{S}' \parallel \mathcal{R}'}{\mathcal{S} \parallel \mathcal{R} \rhd^+ \mathcal{S}' \parallel \mathcal{R}'},$$

$$\frac{\mathcal{S} \parallel \mathcal{R} \rhd \mathcal{S}' \parallel \mathcal{R}' \qquad \mathcal{S}' \parallel \mathcal{R}' \rhd^+ \mathcal{S}'' \parallel \mathcal{R}''}{\mathcal{S} \parallel \mathcal{R} \rhd^+ \mathcal{S}'' \parallel \mathcal{R}''}.$$

Given an initial goal $\Delta \mid \Gamma \vdash_? X : A \parallel \varnothing$, let us observe that the transition process succeeds from this goal exactly when there is a constraint set $\mathcal{R}$ such that $\Delta \mid \Gamma \vdash_? X : A \parallel \varnothing \rhd^+ \bullet \parallel \mathcal{R}$ holds.

Definition 4.2. *A pseudosequent $\Delta \mid \Gamma \vdash_? X : A$ is solvable with respect to a constraint set $\mathcal{Q}$ (or $\mathcal{Q}$-solvable), if there is a constraint $X \approx e \in \mathcal{Q}$ and a program $M = Sol_{\mathcal{Q}}(X \approx e)$ such that the typing $\Delta \mid \Gamma \vdash M : A$ holds.*
Given a constraint set $\mathcal{Q}$ we say that a goal $\mathcal{S} \parallel \mathcal{R}$ is $\mathcal{Q}$-solvable if $\mathcal{R} \subseteq \mathcal{Q}$ and all pseudosequents in $\mathcal{S}$ are $\mathcal{Q}$-solvable.

We remark that the solution is not unique, moreover, there can be an infinite number of solutions. Thus, it should be the human agent who decides the desired solution as she is conducting the whole process. This rules out some relevant inquiries of automated proof-search like the question of the complexity of the proof-search space.

The next lemma guarantees that the solvability of goals is rearward preserved by the transition relation, this characteristic will imply the desired soundness property.

Lemma 1. *Let $\mathcal{Q}$ be a constraint set such that $\mathcal{R}_1, \mathcal{R}_2 \subseteq \mathcal{Q}$. If $\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd^+ \mathcal{S}_2 \parallel \mathcal{R}_2$ and $\mathcal{S}_2 \parallel \mathcal{R}_2$ is solvable with respect to $\mathcal{Q}$ then $\mathcal{S}_1 \parallel \mathcal{R}_1$ is solvable with respect to $\mathcal{Q}$.*

Proof. Let $\mathcal{Q}$ be as required. The proof goes by induction on $\rhd^+$. In the base case we have $\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd \mathcal{S}_2 \parallel \mathcal{R}_2$ and proceed by a nested induction on $\rhd$. We give some cases as example. The remaining are analogous:

— Case (`intro` x): We have $\Delta \mid \Gamma \vdash_? X : A \to B \parallel \mathcal{R} \rhd \Delta \mid \Gamma, x : A \vdash_? Y : B \parallel \mathcal{R}, X \approx \lambda x.Y$. Let us assume that $M = Sol_{\mathcal{Q}}(Y \approx N)$ with $\Delta \mid \Gamma, x : A \vdash M : B$. In this case we have $\Delta \mid \Gamma \vdash (\lambda x.Y)[Y/M] : A \to B$ by rule $(\to R)$. Noting that $\lambda x.M$ is a program, for so is M, and that $\lambda x.M \equiv (\lambda x.Y)[Y/M]$ we get that $\lambda x.M = Sol_{\mathcal{Q}}(X \approx \lambda x.Y)$. Thus the goal $\Delta \mid \Gamma \vdash_? X : A \to B \parallel \mathcal{R}$ is solvable with respect to $\mathcal{Q}$ and the case is done.

— Case (`vassert A :`): In this case we have $\Delta \mid \Gamma \vdash_? X : C \parallel \mathcal{R} \rhd \Delta \mid \cdot \vdash_? X_1 : A \ ;\ \Delta, x : A \mid \Gamma \vdash_? X_2 : C \parallel \mathcal{R}, X \approx \mathsf{elet}(X_1, x.X_2)$. Let us assume that $\Delta \mid \cdot \vdash M_1 : A$ and $\Delta, x : A \mid \Gamma \vdash M_2 : C$ where M_1, M_2 are programs such that there

are equations $Y_1 \approx N_1, Y_2 \approx N_2 \in \mathcal{Q}$ with $M_1 = Sol_{\mathcal{Q}}(Y_1 \approx N_1), M_2 = Sol_{\mathcal{Q}}(Y_1 \approx N_2)$. From the above typings we get, by the (SUBSTE) rule, that $\Delta \mid \Gamma \vdash \mathsf{elet}(M_1, x.M_2) : C$. Finally we observe that $\mathsf{elet}(M_1, x.M_2) = Sol_{\mathcal{Q}}(X \approx \mathsf{elet}(X_1, x.X_2))$. Thus the goal $\Delta \mid \Gamma \vdash_? X : C \parallel \mathcal{R}$ is solvable with respect to $\mathcal{Q}$ as desired.

— Case (seq): We have here that there exists a sequence $\mathcal{S}$ such that $\mathcal{S}_1; \mathcal{S} \parallel \mathcal{R}_1 \rhd \mathcal{S}_2; \mathcal{S} \parallel \mathcal{R}_2$ where $\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd \mathcal{S}_2 \parallel \mathcal{R}_2$. Let us assume that $\mathcal{S}_2; \mathcal{S} \parallel \mathcal{R}_2$ is solvable with respect to $\mathcal{Q}$. This implies in particular that $\mathcal{S}_2 \parallel \mathcal{R}_2$ is $\mathcal{Q}$-solvable, from which the nested induction hypothesis now yields that $\mathcal{S}_1 \parallel \mathcal{R}_1$ is $\mathcal{Q}$-solvable. From this we can conclude that the sequence $\mathcal{S}_1; \mathcal{S}$ is $\mathcal{Q}$-solvable (observe that the part $\mathcal{S}$ was already $\mathcal{Q}$-solvable due to the original assumption). Thus, the goal $\mathcal{S}_1; \mathcal{S} \parallel \mathcal{R}_1$ is solvable with respect to $\mathcal{Q}$, as desired. This finishes the nested induction that proves the base case.

For the inductive step, we have that $\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd^+ \mathcal{S}_2 \parallel \mathcal{R}_2$ where there is a goal $\mathcal{S}_3 \parallel \mathcal{R}_3$ such that $\mathcal{S}_1 \parallel \mathcal{R}_1 \rhd \mathcal{S}_3 \parallel \mathcal{R}_3$ and $\mathcal{S}_3 \parallel \mathcal{R}_3 \rhd^+ \mathcal{S}_2 \parallel \mathcal{R}_2$. Assuming that $\mathcal{S}_2 \parallel \mathcal{R}_2$ is solvable with respect to $\mathcal{Q}$ the I.H. yields $\mathcal{S}_3 \parallel \mathcal{R}_3$ is solvable with respect to $\mathcal{Q}$, the already proved base case now yields that $\mathcal{S}_1 \parallel \mathcal{R}_1$ is solvable with respect to $\mathcal{Q}$, as desired. □

Theorem 4.1 (Soundness of the synthesis process)**.** *Let $\Delta \mid \Gamma \vdash_? X : A$ be a synthesis problem. If $\Delta \mid \Gamma \vdash_? X : A \parallel \varnothing \rhd^+ \bullet \parallel \mathcal{R}$ then there is a program M such that $\Delta \mid \Gamma \vdash M : A$.*

Proof. It is clear that the goal $\bullet \parallel \mathcal{R}$ is solvable with respect to $\mathcal{R}$, hence, the Lemma 1 yields that $\Delta \mid \Gamma \vdash_? X : A \parallel \varnothing$ is also solvable with respect to $\mathcal{R}$. This fact ensures the existence of the desired program M. □

According to this theorem our type-based synthesis process is correct. This ends our exposition. Let us finish this paper with some remarks.

5 Final Remarks

In this paper we presented a dual-context sequent calculus $\mathcal{GS}4$ for the necessity fragment of the constructive modal logic S4, originated in our previous work [19], as a type system for lambda-terms. The modal types allow us to make a distinction between values with essentially the same functionality, namely ordinary values (inhabiting the type A) and enhanced values (inhabiting the type $\Box A$). This distinction is required by several applications.

The specific left rules for implication and necessity as well as the dual-context feature enable us to define a simple and intuitive sound bottom-up synthesis process involving a left-to-right depth-first proof-search, which, with the help of constraint-sets and the backward reading of the typing rules succeeds in returning a correct-by-construction modal lambda-term.

The procedure is human-driven, in the sense of modern interactive theorem provers, a feature that allows to define the synthesis process without technical modifications of the inference rules, unlike some proposals of automated proof-search [29, 1, 26].

Towards a more realistic programming environment we intend to extend the current approach to a dual-context sequent calculus for the full modal logic S4, related to our work in [12]. Another important task is to extend the here presented results to the classical version of S4. This requires a handling of classical negation suitable for proof-search, in particular the use of a traditional multi conclusion sequent calculus is not convenient.

Some other programming language features, like a detailed study of the operational semantics and the extension of $\mathcal{GS}4$ with recursion and memory references in the lines of [22, 8], have to be integrated in this quest.

Another important research topic consists of mechanizing[5] the current results, following our previous work [11]. With respect to other approaches of type-based synthesis in modal logic, we consider important to relate our approach with

[5] This is the main reason for defining contexts as lists instead of sets or multisets.

those involving intersection types, like [14, 9]. This would complicate the constraints, due to the fact that the typing rules for intersection are obviously not syntax-directed. This happens also in other richer type systems, for instance [5] involving some kind of polymorphism.

In some cases the constraints might be unsolvable, for example if the constraint-sets are cyclic or contain a recursive constraint. But even if they are solvable, their solutions would certainly require more powerful tools, such as (higher-order) unification.

Acknowledgments

This research is being supported by DGAPA-PAPIIT UNAM grant IN119920.

References

1. **Andrikonis, J. (2012).** Loop-free calculus for modal logic s4. i. Lithuanian Mathematical Journal, Vol. 52, No. 1, pp. 1–12.
2. **Avron, A., Honsell, F., Miculan, M., Paravano, C. (1998).** Encoding Modal Logics in Logical Frameworks. Studia Logica, Vol. 60, No. 1, pp. 161–208.
3. **Barber, A. G., Plotkin, G. (1997).** Dual intuitionistic linear logic. Technical Report LFCS-96-347, University of Edinburgh.
4. **Basin, D., Deville, Y., Flener, P., Hamfelt, A., Fischer Nilsson, J. (2004).** Synthesis of programs in computational logic. In **Bruynooghe, M., Lau, K.-K.**, editors, Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 30–65.
5. **Bessai, J., Dudenhefner, A., Düdder, B., Chen, T.-C., de'Liguoro, U., Rehof, J. (2015).** Mixin Composition Synthesis Based on Intersection Types. **Altenkirch, T.**, editor, 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015), volume 38 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 76–91.
6. **Borghuis, T., Feijs, L. (2000).** A constructive logic for services and information flow in computer networks. The Computer Journal, Vol. 43, No. 4, pp. 274–289.
7. The Coq Development Team **(2020).** The Coq Proof Assistant Reference Manual Version 8.11.
8. **Davies, R., Pfenning, F. (2001).** A modal analysis of staged computation. J. ACM, Vol. 48, No. 3, pp. 555–604.
9. **Düdder, B., Martens, M., Rehof, J. (2014).** Staged composition synthesis. **Shao, Z.**, editor, Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 67–86.
10. **Fitting, M. C. (1983).** Proof Methods for Modal and Intuitionistic Logics. Synthese Library. Springer Netherlands.
11. **González-Huesca, L., Miranda-Perea, F. E., Linares-Arévalo, P. S. (2019).** Axiomatic and dual systems for constructive necessity, a formally verified equivalence. Journal of Applied Non-Classical Logics, Vol. 29, No. 3, pp. 255–287.
12. **González Huesca, L. d. C., Miranda-Perea, F. E., Linares-Arévalo, P. S. (2020).** Dual and Axiomatic Systems for Constructive S4, a Formally Verified Equivalence. Electronic Notes in Theoretical Computer Science, Vol. 348, pp. 61 – 83. 14th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2019).
13. **Heilala, S., Pientka, B. (2007).** Bidirectional decision procedures for the intuitionistic propositional modal logic is4. Proceedings of the 21st international conference on Automated Deduction: Automated Deduction, CADE-21, Springer-Verlag, Berlin, Heidelberg, pp. 116–131.
14. **Henglein, F., Rehof, J. (2016).** Modal intersection types, two-level languages, and staged synthesis. In **Probst, C., Hankin, C., Hansen, R.**, editors, Semantics, logics, and calculi, Lecture notes in computer science. Springer, pp. 289–312. Nielsons' Festschrift.
15. **Hudelmaier, J. (1996).** A contraction-free sequent calculus for s4. In **Wansing, H.**, editor, Proof Theory of Modal Logic, Applied Logic Series. Springer Netherlands, pp. 3–15.
16. **Kavvos, G. A. (2017).** Dual-context calculi for modal logic. 32nd Annual ACM/IEEE Symposium

on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017, pp. 1–12.

17. **Martin-Löf, P. (1996).** On the meanings of the logical constants and the justifications of the logical laws. Nordic J. Philos. Logic, Vol. 1, No. 1, pp. 11–60.
18. **Mints, G., Orevkov, V., Tammet, T. (1996).** Transfer of sequent calculus strategies to resolution for s4. In **Wansing, H.**, editor, Proof Theory of Modal Logic, Applied Logic Series. Springer Netherlands, pp. 17–31.
19. **Miranda-Perea, F. E., del Carmen González Huesca, L., Linares-Arévalo, P. S. (2020).** On interactive proof-search for constructive modal necessity. Electronic Notes in Theoretical Computer Science, Vol. 354, pp. 107 – 127. Proceedings of the Eleventh and Twelfth Latin American Workshop on Logic/Languages, Algorithms and New Methods of Reasoning (LANMR).
20. **Miranda-Perea, F. E., Linares-Arévalo, P. S., Aliseda-Llera, A. (2015).** How to prove it in natural deduction: A tactical approach. CoRR, Vol. abs/1507.03678.
21. **Miyamoto, K., Igarashi, A. (2004).** A modal foundation for secure information flow. **Sabelfeld, A.**, editor, Workshop on Foundations of Computer Security, pp. 187–203.
22. **Moody, J. (2004).** Logical mobility and locality types. **S., E.**, editor, Proceedings of the 14th International Conference on Logic Based Program Synthesis and Transformation. LOPSTR 2004, volume 3573 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, pp. 69–84.
23. **Murphy VII, T., Crary, K., Harper, R., Pfenning, F. (2004).** A symmetric modal lambda calculus for distributed computing. Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, LICS '04, IEEE Computer Society, Washington, DC, USA, pp. 286–295.
24. **Pfenning, F., Davies, R. (2001).** A judgmental reconstruction of modal logic. Mathematical Structures in Comp. Sci., Vol. 11, No. 4, pp. 511–540.
25. **Pfenning, F., Wong, H. (1995).** On a modal lambda calculus for S4. **Brookes, S. D., Main, M. G., Melton, A., Mislove, M. W.**, editors, Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, LA, USA, March 29 - April 1, 1995, volume 1 of Electronic Notes in Theoretical Computer Science, Elsevier, pp. 515–534.
26. **Pliuškevičius, R., Pliuškevičienė, A. (2008).** A new method to obtain termination in backward proof search for modal logic s4. Journal of Logic and Computation, Vol. 20, No. 1, pp. 353–379.
27. **Poggiolesi, F. (2010).** Gentzen Calculi for Modal Propositional Logic. Trends in Logic. Springer Netherlands.
28. **Schroeder-Heister, P. (2011).** Implications-as-Rules vs. Implications-as-Links: An Alternative Implication-Left Schema for the Sequent Calculus. J. Philosophical Logic, Vol. 40, No. 1, pp. 95–101.
29. **Stone, M. (2005).** Disjunction and modular goal-directed proof search. ACM Trans. Comput. Logic, Vol. 6, No. 3, pp. 539–577.

Article received on 09/10/2020; accepted on 11/02/2021.
Corresponding author is Favio E. Miranda-Perea.