# Towards a Simply Typed C$\mathcal{AL}$culus for Semantic Knowledge Bases*

Michael Mendler, Stephan Scheele
Informatics Theory Group
University of Bamberg, Germany
{michael.mendler,stephan.scheele}@uni-bamberg.de

**Abstract**

This paper demonstrates how a constructive version of the description logic $\mathcal{ALC}$ can serve as a semantic type system for an extension of the simply typed $\lambda$-calculus to express computations in knowledge bases. This $c\mathcal{AL}$culus embodies a functional core language which provides static type checking of semantic information processing of data whose structure is organised under a relational data model as used in description logics. The $c\mathcal{AL}$culus arises from a natural interpretation of the tableau rules for constructive $\mathcal{ALC}$ following the Curry-Howard-Isomorphism.

## 1 Introduction

Description logics (DL) are specification formalisms, which have found numerous applications in the semantic processing of data. DLs [2] are a family of knowledge representation languages that can be used to specify the terminological knowledge of a specific domain in a structured and formally well-understood way. DLs are used in the field of semantic data bases, in applications of the Semantic Web and as formal grounding for the W3C-endorsed Web Ontology Language (OWL). Knowledge is expressed in terms of a set of concepts and roles which specify a terminological component called TBox, i.e., a controlled vocabulary about a specific domain. This vocabulary can be associated with a set of facts/assertions which is called ABox, combined they build up a knowledge base.

A fairly recent idea is to employ DLs as programming language type systems [15, 17, 23]. Our work is aimed at programming with/in ABoxes as data structures. Usual ABox reasoning corresponds to type checking and TBox reasoning is programming. In this way, DLs may be used as a static specification formalism for programming in knowledge bases. In this paper we look at the use of DLs as a *programming type system* (see e.g., [20]) which naturally requires a constructive setting in contrast to the standard classical semantics of DLs. Specifically, we show how the constructive description logic $c\mathcal{ALC}$, introduced in [15], may be turned into a type system for ABox data streams much in the spirit of typed functional programming. The general benefits of the Curry-Howard Isomorphism (*proofs-as-computations*) in DLs have been argued in [5, 4].

## 2 Syntax and Semantics of $c\mathcal{ALC}$ Types

The main syntactic building blocks of description logics are concepts (classes), representing a class of objects, roles (properties) that are relating objects and entities (individuals) which represent specific objects. From atomic concepts like $Frog$ composite concepts can be constructed

*Research project funded by the German Research Foundation (DFG) under ME 1427/4-1.

using concept constructors, e.g. $Frog \sqcap \exists hasColor.Green$ specifies an object which is in $Frog$ and that is related through the $hasColor$ role with an object from the concept $Green$. A subsumption relationship represents that a concept is more general than another one, e.g. $Frog$ is subsumed by the concept $Animal$. A DL knowledge base usually consists of a TBox and an ABox. The TBox is a set of axioms stating general properties of concepts and roles which form the structure of allowed worlds. In the most general case terminological axioms are formed by inclusions $C \sqsubseteq D$ and equivalences $C \equiv D$, where $C$ and $D$ are concepts. The ABox comprises assertions on individual objects and thereby defines the structure of a particular world, for instance the statement that Kermit is a $Frog$ that has the color $Lime$ can be stated by the assertions $Frog(Kermit)$ and $hasColor(Kermit, Lime)$.

Just like the types NAT or BOOL specify the structure and semantical meaning of data structures in ordinary programming languages, the logic $\mathcal{ALC}$ may be used as a type language for programming in structured, entity relationship-alike semantic knowledge bases. At the outset this implies two extensions in relation to standard $\mathcal{ALC}$ [2]. First, the classical two-valued semantics need to be replaced by an intuitionistic many-valued interpretation. Second subsumption, which types functional computations, becomes a first-order binary operator of the language, to give a higher-order functional programming language. Such an extension of $\mathcal{ALC}$, called $c\mathcal{ALC}$ has been introduced in [15, 16].

*Concept descriptions* in $c\mathcal{ALC}$ are based on sets of *role names* $N_R$ and *concept names* $N_C$ and formed as follows, where $A \in N_C$ and $R \in N_R$:

$$C, D \quad \rightarrow \quad A \mid \bot \mid C \sqcap D \mid C \sqcup D \mid C \sqsubseteq D \mid \exists R.C \mid \forall R.C.$$

Being a first class operator, subsumption can be nested arbitrarily as in $((A \sqsubseteq C) \sqsubseteq B) \sqsubseteq D$, which is crucial to type higher-order programs. We use $C \equiv D$ as an abbreviation of $(C \sqsubseteq D) \sqcap (D \sqsubseteq C)$. Constructive negation can be coded $\neg C = C \sqsubseteq \bot$.

Perhaps not surprisingly, negation plays a secondary role for programming type systems and will not be discussed much in this paper other than as a modifier of atomic concept descriptions. The computational meaning of general negated types can be related to continuation-style programming [1] or backtracking [11]. It is likely to be complex to carry this over to DL-programming, however, and deserves its own technical development that we will not attempt to tackle here.

To begin with, we recall the definition of a constructive $\mathcal{ALC}$ interpretation from [15]:

**Definition 1.** A *constructive interpretation* or *constructive model* of $c\mathcal{ALC}$ is a structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \preceq^{\mathcal{I}}, \bot^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of

- a non-empty set $\Delta^{\mathcal{I}}$ of *entities*, the universe of discourse in which each entity represents a partially defined, or abstract individual;
- a *refinement* pre-ordering $\preceq^{\mathcal{I}}$ on $\Delta^{\mathcal{I}}$, i.e., a reflexive and transitive relation;
- an interpretation function $\cdot^{\mathcal{I}}$ mapping each role name $R \in N_R$ to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and each atomic concept $A \in N_C$ to a set $\bot^{\mathcal{I}} \subseteq A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ which is closed under refinement, i.e., $x \in A^{\mathcal{I}}$ and $x \preceq^{\mathcal{I}} y$ implies $y \in A^{\mathcal{I}}$. If $x\, R^{\mathcal{I}}\, z$ then $z$ is called filler of $R$ for $x$. We also write $(x, z) \in R^{\mathcal{I}}$ or in fact $x\, R\, z$ if $\mathcal{I}$ is understood;
- finally a subset $\bot^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ of *fallible* entities closed under refinement and role filling, i.e., $x \in \bot^{\mathcal{I}}$ and $x \preceq^{\mathcal{I}} y$ implies $y \in \bot^{\mathcal{I}}$, for every fallible entity $x$ exists a fallible filler, i.e., $\forall R.\, \exists z.\, x\, R^{\mathcal{I}}\, z \;\&\; z \in \bot^{\mathcal{I}}$ and all fillers of a fallible entity $x$ are fallible, i.e. $\forall R.\, \forall z.\, x\, R^{\mathcal{I}}\, z \Rightarrow z \in \bot^{\mathcal{I}}$. □

Constructive models $\mathcal{I}$ extend the classical models for $\mathcal{ALC}$ by a pre–ordering $\preceq^{\mathcal{I}}$ for capturing possible refinement between entities and by a notion of fallible entities $\bot^{\mathcal{I}}$ for interpreting empty computations. The refinement relation $x \preceq y$ may be used to capture a number of important special semantical dimensions uniformly without invoking additional syntactic overhead:

- $x$ is a stream of objects arising by serialisation of a data base table or from other continuous data production processes, e.g. sensor networks. In this case, $x = x_1 \cdot x_2 \cdot \ldots$ may be a finite or an infinite stream where $x \preceq y$ is the suffix ordering. E.g., $x$ may be an object about which information only arrives peu a peu or that is available only in a cumulative way through repeated accesses.
- $x$ is an abstraction of data records. Each refinement $y$ of $x$ has all the attributes of $x$ and on those the same values, but possibly also additional attribute dimensions. E.g., $x$ may be the result of suppressing information in an attempt to optimise calculations on a large data base. Every application of a projection on a data base table creates an abstraction in this sense.

Fallible elements $b \in \bot^{\mathcal{I}}$ may be thought of as over-constrained tokens of information, self-contradictory objects of evidence or undefined computations. E.g., they may be used to model the situation in which computing a role-filler for an abstract individual fails.

We obtain the standard classical models of $\mathcal{ALC}$ whenever $\preceq^{\mathcal{I}}$ is the identity relation and $\bot^{\mathcal{I}}$ is empty. Generally speaking, $\preceq$ internalises the Open World Assumption locally for every object of an interpretation. Therefore, we can identify a constructive model with an ABox. A constructive model is a network of structurally interrelated data contexts which may be very large and distributed. Concept descriptions express structural invariants in these semantical networks. They can be used as type specifications for a programming language to manipulate and compute such data coherently. The following definition establishes the basic validity relationship between data and types:

**Definition 2.** Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \preceq^{\mathcal{I}}, \bot^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be a constructive model. The interpretation $\mathcal{I}$ is lifted from atomic $\bot$, $A$ to arbitrary concepts as follows, where $\Delta^{\mathcal{I}}_c =_{df} \Delta^{\mathcal{I}} \setminus \bot^{\mathcal{I}}$ is the set of *non-fallible* elements in $\mathcal{I}$:

$$\begin{array}{rcl}
\top^{\mathcal{I}} & =_{df} & \Delta^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} & =_{df} & \{x \mid \forall y \in \Delta^{\mathcal{I}}_c.\ x \preceq^{\mathcal{I}} y \Rightarrow y \notin C^{\mathcal{I}}\} \\
(C \sqcap D)^{\mathcal{I}} & =_{df} & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} & =_{df} & C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(C \sqsubseteq D)^{\mathcal{I}} & =_{df} & \{x \mid \forall y \in \Delta^{\mathcal{I}}_c.\ (x \preceq^{\mathcal{I}} y \ \&\ y \in C^{\mathcal{I}}) \Rightarrow y \in D^{\mathcal{I}}\} \\
(\exists R.C)^{\mathcal{I}} & =_{df} & \{x \mid \forall y \in \Delta^{\mathcal{I}}_c.\ x \preceq^{\mathcal{I}} y \Rightarrow \exists z \in \Delta^{\mathcal{I}}.\ (y,z) \in R^{\mathcal{I}} \ \&\ z \in C^{\mathcal{I}}\} \\
(\forall R.C)^{\mathcal{I}} & =_{df} & \{x \mid \forall y \in \Delta^{\mathcal{I}}_c.\ x \preceq^{\mathcal{I}} y \Rightarrow \forall z \in \Delta^{\mathcal{I}}.\ (y,z) \in R^{\mathcal{I}} \Rightarrow z \in C^{\mathcal{I}}\}.
\end{array}$$

We will write $\mathcal{I}; x \models C$ as an abbreviation for $x \in C^{\mathcal{I}}$. □

**Example 1.** The recursive concept definition $\textsc{Tree} \equiv (\exists \mathit{leaf}.\textsc{nat}) \sqcup \exists \mathit{node}.\textsc{Tree} \sqcap \textsc{Tree}$ specifies an ABox data structure of binary trees with natural numbers stored in leaves. □

**Example 2.** As mentioned before, c$\mathcal{ALC}$ can be used to specify the type of potentially infinite streams $x = x_1 \cdot x_2 \cdot \ldots$. Under interpretation $\mathcal{I}$ let NAT and BOOL be the atomic concepts of natural numbers and booleans, i.e., such that $\textsc{nat}^{\mathcal{I}} = \mathbb{N}$ and $\textsc{bool}^{\mathcal{I}} = \mathbb{B}$. We assume there is an indicated (functional) role *val* that relates a stream with its first data entity considered as an infinite constant stream, if such exists and the empty stream ($\epsilon$) otherwise. More concretely

$val(\epsilon, \epsilon)$ and $val(v \cdot s, v^\infty)$, and for instance with concrete streams, $val((2, \mathtt{T}) \cdot \mathtt{T} \cdot \mathtt{F}, (2, \mathtt{T})^\infty)$ and $val(\mathtt{T} \cdot \mathtt{F}, \mathtt{T}^\infty)$.

Let $s_0 = (0, \mathtt{T}) \cdot (1, \mathtt{T}) \cdot (1, \mathtt{F}) \cdot \ldots$ be a stream of pairs of naturals and booleans. Then, $s_0$ satisfies the concept $\exists val.(\text{NAT} \sqcap \text{BOOL})$. De-multiplexing of $s_0$ is a function that is flattening the pairs of naturals and booleans into a stream which is oscillating between values of type NAT and those of type BOOL. This function applied to $s_0$ results in the stream $s_1 = 0 \cdot \mathtt{T} \cdot 1 \cdot \mathtt{T} \cdot 1 \cdot \mathtt{F} \cdots$. The oscillation between NAT and BOOL can be specified by the concept $Osc =_{df} \neg\exists val.\text{NAT} \sqcap \neg\exists val.\text{BOOL} \sqcap \forall val.(\text{NAT} \sqcup \text{BOOL})$ which is only possible in constructive logic. The classical meaning of $Osc$, in classical $\mathcal{ALC}$, is the empty set. Under the Curry-Howard Isomorphism [26, 25] the product type $C \times D$ is the constructive interpretation of conjunction $C \sqcap D$ and function spaces $C \to D$ are the constructive reading of subsumptions $C \sqsubseteq D$.
In this regard, multiplexing and de-multiplexing of the above described data streams would be different constructive realisations of the following subsumptions:

$$\exists val.(\text{NAT} \sqcap \text{BOOL}) \sqsubseteq \exists val.(\text{NAT} \sqcup \text{BOOL}) \qquad \exists val.(\text{NAT} \sqcup \text{BOOL}) \sqsubseteq \exists val.(\text{NAT} \sqcap \text{BOOL}).$$

Note for $\exists val.(\text{NAT} \sqcup \text{BOOL})$ that the use of $\exists val$ performs the segmentation of the stream such that the concept NAT $\sqcup$ BOOL is executed element-wise rather than globally. This illustrates that the classical principle distribution of existential $\exists$ over disjunction $\sqcup$, i.e., the equivalence $\exists val.(\text{NAT} \sqcup \text{BOOL}) \equiv \exists val.\text{NAT} \sqcup \exists val.\text{BOOL}$ does not hold in $c\mathcal{ALC}$. The stream $s_1$ satisfies the concept $\exists val.(\text{NAT} \sqcup \text{BOOL})$ saying that the first element of each suffix sequence is of type NAT or BOOL. But $s_1$ does not satisfy the concept $\exists val.\text{NAT} \sqcup \exists val.\text{BOOL}$ since this would require that all elements of $s_1$ are either NAT or all are BOOL. □

The language $c\mathcal{ALC}$ is related to the constructive modal logic CK [14, 27, 3, 13] as $\mathcal{ALC}$ is related to the classical modal logic K. $c\mathcal{ALC}$ is the multimodal version of CK where $\forall R$ is the multimodal (or indexed) version of the box modality $\Box$ and $\exists R$ is the indexed version of the diamond $\Diamond$. Note that CK is not the intuitionistic analogue to classical K in the sense of Fischer-Servi and Simpson [24], i.e. adding of the axiom of the Excluded Middle $C \vee \neg C \equiv \top$ does not collapse the theory to classical $K$. Instead CK can be seen as a generalisation of the logical system of intuitionistic K. We discuss their relation as follows.

It is traditional in intuitionistic modal logics to dualise $\Box$ as a monotonic $\wedge$-preserving operator and to define $\Diamond$ as a monotonic $\vee$-preserving modality. There are two equivalent axiomatisations of this idea, Plotkin and Stirling's IK [22, 24] and Fischer-Servi's system [7], called FS in [8]. The former is given by the following axioms and rules:

| **Axioms** (IK) | | **Rules** (IK) |
|---|---|---|
| All theorems of IPL | IK3 : $\neg\Diamond\bot$ | MP : $A$ and $A \supset B$ implies $B$ |
| IK1 : $\Box(A \supset B) \supset (\Box A \supset \Box B)$ | IK4 : $\Diamond(A \vee B) \supset (\Diamond A \vee \Diamond B)$ | Nec : $A$ implies $\Box A$ |
| IK2 : $\Box(A \supset B) \supset (\Diamond A \supset \Diamond B)$ | IK5 : $(\Diamond A \supset \Box B) \supset \Box(A \supset B)$ | |

Like classical K, the logic IK/FS admits of an elementary Kripke style model theory. The systems IK/FS arise from the standard intuitionistic semantics of the propositional connectives and the interpretation of $\Box$, $\Diamond$ as universal and existential quantifiers over accessible worlds in an intuitionistic meta-theory:

$$x \models \Box C \quad \textit{iff} \quad \forall y.\, x \preceq y \Rightarrow \forall z.\, y\, R\, z \Rightarrow z \models C \tag{1}$$

$$x \models \Diamond C \quad \textit{iff} \quad \exists z.\, x\, R\, z \;\&\; z \models C \tag{2}$$

In intuitionistic logic all propositions must be closed under refinement, i.e., $x \models C$ and $x \preceq y$ implies $y \models C$. For this to hold under definition (2) of $\Diamond$, the models of IK/FS need to satisfy

confluence between $\preceq$ and $R$, i.e., the frame condition $\preceq^{-1}; R \subseteq R; \preceq^{-1}$ where ; denotes composition of relations. The modal logic CK, which is the basis for $c\mathcal{ALC}$ does not depend on such frame conditions. CK is a sub-theory of IK/ FS which does not contain the axioms IK3, IK4, IK5. They are problematic from a constructive point of view as argued in [14].

It seems to us that in computational type theories [12, 6] or modal type theories [10, 21, 18, 19], where constructive proofs turn into $\lambda$-programs, the schemes IK3 – IK5 fail to have a uniform computational justification. On the other hand, the schemes that do appear to be computationally justified are IK1 and IK2. Restricting to these axioms yields the constructive system known as CK [14, 3, 13].

# 3 The Typed Language $\lambda_{\mathcal{ALC}}$

Our language $\lambda_{\mathcal{ALC}}$ extends the simply-typed $\lambda$-calculus with explicit pairing and disjoint sums [9] by constructions to handle roles and fillers. Like in object-oriented programming we distinguish between filler objects and filler methods. The former are references to local data bases or data contexts and the latter are computational methods that can be executed on these data bases. Both together constitute what we might call a *computational knowledge base*.
The difference to standard $\lambda$-calculus is that computations are only available as local methods by reference to knowledge bases reified as role fillers. By way of role filling we can change the knowledge base in context and jump between those. Role filling can be understood as a computational process where we distinguish between input and output filling actions. This gives a rather natural operational interpretation for the duality between existential and universal role filling. Specifically, for navigation the language provides explicit path expressions of the shape $R!k.e$ and $R?a.e$. These resemble a process-algebraic prefix notation suggesting an implicit *interaction* relation of the form

$$R!k.e \xrightarrow{R!k} e \qquad \text{or} \qquad R?a.e \xrightarrow{R?a} e$$

where $k$ is a filler (an entry point to a local knowledge base), $a$ a filler variable and $R \in N_R$ a role name. The former represents an *output action* $R!k$ which provides a reference $k$ for an $R$-filler and $e$ is the *filler method* that can be executed in the context of $k$. The second kind of action $R?a$ accepts an $R$-filler $a$ from the environment and executes method $e$ there. The distinction between input $R?a$ and output actions $R!k$ is reflected in the difference between universal and existential quantification of our $c\mathcal{ALC}$ types. Existentials $\exists R.C$ specify output actions and universals $\forall R.C$ pertain to input. A method $p$ of type $\exists R.C$ contains computations of $R$-fillers of type $C$ which are offered at a specific location determined by $p$. In contrast, a method $p$ typed $\forall R.C$ generates a computation of type $C$ at *any* $R$-filler determined by the environment of $p$.

## 3.1 Syntax of $\lambda_{\mathcal{ALC}}$

The syntax is as seen in Fig. 1. We only explain the parts that extend the $\lambda$-calculus. The expression $\texttt{let}\, R!a.x = e_1 \,\texttt{in}\, e_2$ takes an $R$-filler expression $e_1$ and matches it against the pattern $R!a.x$ which binds the filler object to variable $a$ and the filler methods to variable $x$. The expression $R!k.e$ packages up a filler $k$ with a particular method $e$. Formally, $R!$ is the constructor and `let` the destructor for objects of type $\exists R.C$. For the concept type $\forall R.C$ the constructor is the abstraction $R?$ and the destructor is application @. As explained above, these work analogously to function abstraction and function application, respectively.
There are several variable binding operators in this language:

| | | | |
|---|---|---|---|
| $e$ | $::=$ | $x$ | value *variable* |
| | $\mid$ | $a$ | filler variable |
| | $\mid$ | $k$ | filler object expression/reference |
| | $\mid$ | $(e, e)$ | pairs |
| | $\mid$ | $\pi_i\, e$ | projections $i = 1, 2$ |
| | $\mid$ | $\texttt{case}\ e\ \texttt{of}\ [\iota_1\, x_1 \to e \mid \iota_2\, x_2 \to e]$ | case analysis |
| | $\mid$ | $\iota_i\, e$ | injection $i = 1, 2$ |
| | $\mid$ | $e\, e$ | function application |
| | $\mid$ | $\lambda x.\, e$ | function abstraction |
| | $\mid$ | $\texttt{let}\ R!a.x = e\ \texttt{in}\ e$ | filler opening |
| | $\mid$ | $R!k.e$ | filler closing |
| | $\mid$ | $R?a.e$ | filler abstraction |
| | $\mid$ | $e@k$ | filler instantiation |

Figure 1: Syntax of $\lambda_{\mathcal{ALC}}$.

- $\lambda x.\, e$ binds variable $x$ in $e$,
- $\texttt{case}\ e\ \texttt{of}\ [\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2]$ binds variables $x_1$, $x_2$ in $e_1$, $e_1$, respectively,
- $\texttt{let}\ R!a.x = e_1\ \texttt{in}\ e_2$ binds both value variable $x$ and filler variable $a$ in $e_2$,
- $R?a.e$ binds filler variable $a$ in $e$.

As usual an expression $e$ is called *closed* if all occurrences of variables in $e$ are in the scope of an associated binder. Variables which are not bound are called *free* in $e$. We will identify expressions up to renaming of bound variables. Syntactic substitution $e_1\{e_2/x\}$ is assumed to be capture-avoiding, i.e., automatically renaming bound variables of $e_1$ to prevent conflicts with free variables of $e_2$.

## 3.2 Typing for $\lambda_{\mathcal{ALC}}$

Now we show how concept descriptions from $c\mathcal{ALC}$ can be attached as static typing information to specify contextually well-formed or "well-localised" $\lambda_{\mathcal{ALC}}$-terms. This is consistent with the model-theoretic semantics of Sec. 2 in the sense of Curry-Howard (propositions-as-types, proofs-as-programs), viz. that closed $\lambda_{\mathcal{ALC}}$-terms exist for exactly those types that are theorems of $c\mathcal{ALC}$. Thus, $\lambda_{\mathcal{ALC}}$ plays the same role for the constructive logic and model-theory of $c\mathcal{ALC}$ [15] that simply-typed $\lambda$-calculus does for intuitionistic propositional logic. Every proof in the typing system for $c\mathcal{ALC}$ corresponds to the construction of a well-localised $\lambda_{\mathcal{ALC}}$ program.

The typing system uses typing judgements, also called *sequents*, of the form $\Sigma \vdash \Psi$ in which $\Sigma$ is a *typing context* containing typing assumptions and $\Psi$ is the actual *typing statement* which gives typing guarantees on the basis of assumptions $\Sigma$. Context and statement are sequences of sets in which the information is localised to individual filler objects connected by filler relations. Specifically, a context in general looks like

$$\Sigma \;=\; \Gamma_1 \gg_{S_2?b_2} \Gamma_2 \gg_{S_3?b_3} \cdots \gg_{S_f?b_f} \centerdot \Gamma_f \gg_{S_{f+1}?b_{f+1}} \cdots \gg_{S_n?b_n} \Gamma_n \qquad (3)$$

for role names $S_i \in N_R$ and filler variables $b_i$ $(i = 1, \ldots, n)$. Each $\Gamma_i$ is a set of local typing assumptions, or filler *scopes*

$$\Gamma_i \;=\; \{t_{i1} : C_{i1}, t_{i2} : C_{i2}, \ldots, t_{im_i} : C_{im_i}\} \tag{4}$$

for some (destructor) terms $t_{ij}$ and types $C_{ij}$. The context $\Sigma$ in (3) represents the assumption that all objects specified in $\Gamma_i$ are available as filler methods of an $S_i$-filler referred to by variable $b_i$ of the object at level $\Gamma_{i-1}$. The first scope $\Gamma_1$ is referred to as the *root* and the last $\Gamma_n$ as the *leaf* assumption. As can be seen in (3) one of these local assumptions $\centerdot\,\Gamma_f$, is distinguished by a *focus marker*. The focus $\centerdot$ highlights the filler object $b_f$ in $\Sigma$ relative to which the typing guarantees $\Psi$ in sequent $\Sigma \vdash \Psi$ are made. The number of filler steps to the right of the focus is called the *length* of a context $\Sigma$ and the number of contexts before the focus is called its *depth*. E.g., the context $\Sigma$ in (3) has depth $f-1$ and length $n-f$. Thus, a context with zero depth and zero length is a single set $\Sigma = \centerdot\,\Gamma$.

The typing statement $\Psi$, too, is split into a sequence of guarantees

$$\Psi \;=\; \Phi_1 \gg_{R_2!k_2} \Phi_2 \gg_{R_3!k_3} \Phi_3 \gg_{R_4!k_4} \cdots \gg_{R_m!k_m} \Phi_m \tag{5}$$

with $R_j \in N_R$ and filler expressions $k_j$ $(j = 1, \ldots, m)$. In contrast to the typing assumptions in $\Sigma$ for which filler relationships are context assumptions and therefore input actions $S_i?b_i$, here we are looking at output actions $R_j!k_j$ which guarantee the existence of filler objects $k_j$. Notice, there is no focus marker in $\Psi$. It is implicit and fixed to the first set $\Phi_1$. Accordingly, the statement $\Psi$ has no depth but length $m-1$. It suffices to consider typing statements of length $\leq 1$ which contain exactly one expression, i.e., sequents $\Sigma \vdash e : D$ or $\Sigma \vdash \emptyset \gg_{R!k} e : D$. On the context side $\Sigma$, restrictions can be made, too, though we do not need to exploit these here.

Before we can present the typing system for $\lambda_{\mathcal{ALC}}$ to derive valid sequents $\Sigma \vdash \Psi$ we need to agree on a couple of meta-level syntactic conventions to handle sequents generically. To begin with, we will treat each element $\Gamma_j$ of the context sequence $\Sigma$ (and similarly for $\Psi$) as an unordered list without duplications, so that if $t : C \in \Gamma_j$ then $\Gamma_j$ is the same as $\Gamma_j, t{:}C$ and $t{:}C, \Gamma_j$. In handling the sequence $\Sigma$ we need to preserve the ordering between the $\Gamma_j$, however. We will usually place the focus at the beginning of the respective local assumption set $\centerdot\,\Gamma_f, t : C$ as done in (3) but sometimes we write $\Gamma_f, t : C\,\centerdot$ to make it appear at the end.

Talking about full context sequences (and analogously for statements $\Psi$) it will be convenient to use associativity of the separators $\gg_{S?b}$ for breaking up a context at any point as in $\Sigma = \Sigma' \gg_{S?b} \Sigma''$ where $\Sigma'$ and $\Sigma''$ are the corresponding sub-sequences. This includes the special case that one of the sub-sequences is empty. E.g., if $\Sigma''$ is empty then $\Sigma' \gg_{S?b} \Sigma'' = \Sigma'$. This is *not* the same as $\Sigma' \gg_{S?b} \emptyset$ keeping in mind the difference between an empty sequence and a singleton sequence consisting of an empty set of assumptions. In a sequent $\Sigma \vdash \Psi$ neither sequence $\Sigma$ nor $\Psi$ will ever be empty, i.e., we have $n, m \geq 1$ in (3) and (5).

We will write $\Sigma, t : C$ to say that the leaf assumption contains $t : C$, i.e., $\Sigma, t : C = \Sigma' \gg_{S?b} \Gamma, t : C$, where $\Sigma'$ is the initial sub-sequence of $\Sigma$ without the leaf set. If $\Sigma'$ is empty, then $\Sigma' \gg_{S?b} \Gamma, t : C$ is the same as the singleton sequence $\Gamma, t : C$. Similarly, $t : C, \Sigma$ means that $t : C$ is in the root, i.e., $t : C, \Sigma = \Gamma, t : C \gg_{S?b} \Sigma'$. Again, $\Sigma'$ may be the empty sequence in which case $\Gamma, t : C \gg_{S?b} \Sigma' = \Gamma, t : C$. Finally, $\Sigma\,\centerdot$ indicates that the focus is in the leaf context.

The typing rules are now given in Figs. 2 and 3 separated into those which deal with the standard $\lambda$-terms and those that are specific to $c\mathcal{ALC}$-types. The former in Fig. 2 are essentially the well-known rules from intuitionistic logic and simply typed $\lambda$-calculus. The rules for modal types in Fig. 3 warrant more detailed explanations and in the following we discuss them one by one.

$$\frac{}{\Sigma \gg_{S?b} \centerdot t : C, \Sigma' \vdash t : C}\, Ax_m$$

$$\frac{\Sigma \gg_{S?b} \centerdot \pi_1\, t : C_1, \pi_2\, t : C_2, \Sigma' \vdash \Psi}{\Sigma \gg_{S?b} \centerdot t : C_1 \sqcap C_2, \Sigma' \vdash \Psi}\, \sqcap L \qquad \frac{\Sigma \vdash e_1 : D_1 \quad \Sigma \vdash e_2 : D_2}{\Sigma \vdash (e_1, e_2) : D_1 \sqcap D_2}\, \sqcap R$$

$$\frac{\Sigma \vdash e : D_1}{\Sigma \vdash \iota_1 e : D_1 \sqcup D_2}\, \sqcup R_1 \qquad \frac{\Sigma \vdash e : D_2}{\Sigma \vdash \iota_2 e : D_1 \sqcup D_2}\, \sqcup R_2$$

$$\frac{\Sigma \gg_{S?b} \centerdot x_1 : C_1, \Sigma' \vdash \Psi_1 \quad \Sigma \gg_{S?b} \centerdot x_2 : C_2, \Sigma' \vdash \Psi_2}{\Sigma \gg_{S?b} \centerdot t : C_1 \sqcup C_2, \Sigma' \vdash \mathtt{case}(t, x_1, x_2, \Psi_1, \Psi_2)}\, \sqcup L$$

$$\frac{\Sigma \gg_{S?b} \centerdot \Sigma' \vdash e_1 : C_1 \quad \Sigma \gg_{S?b} \centerdot t\, e_1 : C_2, \Sigma' \vdash \Psi}{\Sigma \gg_{S?b} \centerdot t : C_1 \sqsubseteq C_2, \Sigma' \vdash \Psi}\, \sqsubseteq L$$

$$\frac{\Sigma \gg_{S?b} \centerdot x : C, \Sigma' \vdash e : D}{\Sigma \gg_{S?b} \centerdot \Sigma' \vdash \lambda x.\, e : C \sqsubseteq D}\, \sqsubseteq R$$

In rule $\sqcup L$ the two right-hand contexts must be both of the same form, i.e., $\Psi_i = e_i : D$ or $\Psi_i = \emptyset \gg_{R!k_i} e_i : D$. In the former case, we define $\mathtt{case}(t, x_1, x_2, \Psi_1, \Psi_2) =_{df} e : D$, in the latter $\mathtt{case}(t, x_1, x_2\Psi_1, \Psi_2) =_{df} \emptyset \gg_{R!k} e : D$, where $e =_{df} \mathtt{case}\ t\ \mathtt{of}\ [\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2]$ and $k =_{df} \mathtt{case}\ t\ \mathtt{of}\ [\iota_1\, x_1 \to k_1 \mid \iota_2\, x_2 \to k_2]$. The variable $x$ must be fresh in $\sqsubseteq R$.

Figure 2: Lambda Typing Rules.

$$\frac{\Sigma \gg_{R?a} \centerdot \Gamma' \vdash e : D}{\Sigma \centerdot \gg_{R?a} \Gamma' \vdash \emptyset \gg_{R!a} e : D}\, Ax_f$$

$$\frac{\Sigma \vdash \emptyset \gg_{R!k} e : D}{\Sigma \vdash R!k.e : \exists R.D}\, \exists R \qquad \frac{\Sigma \centerdot \gg_{R?a} y : C \vdash e : D}{\Sigma, t : \exists R.C \centerdot \vdash \mathtt{let}\ R!a.y = t\ \mathtt{in}\ e : D}\, \exists L$$

$$\frac{\Sigma \gg_{S?b} t@b : C, \Sigma' \vdash \Psi}{\Sigma, t : \forall S.C \gg_{S?b} \Sigma' \vdash \Psi}\, \forall L \qquad \frac{\Sigma \gg_{R?a} \centerdot \emptyset \vdash e : D}{\Sigma \centerdot \vdash R?a.\, e : \forall R.D}\, \forall R$$

The variables $a$ and $y$ must be fresh in rules $\forall R$ and $\exists L$.

Figure 3: Modal Typing Rule.

- The right rule $\exists R$ says that if $e$ represents a filler method of type $D$ for an $R$-filler of the current scope, referred to by $k$, then the output action $R!k.e$ is an object of type $\exists R.D$ at the current focus. Note the difference of length between premise and conclusion: While $e$ in the premise is typed $\vdash \emptyset \gg_{R!k} e : D$, i.e, it is one $R$-step forward from the current scope, the typing $\vdash R!k.e : \exists R.D$ ensures that the closed action $R!k.e$ is available at the current focus rather than somewhere else.

- The purpose of the left introduction rule $\exists L$ is to tell how $R$-filler output actions can be used. As seen in the premise of $\exists L$ we take an expression $e$ of some type $D$ which depends on a $R$-filler variable $a$ and a method $y : C$. These are assumptions about the existence of a filler forward from the current scope, which is why $e$ is typed in the context $\Sigma \centerdot \gg_{R?a} y : C$. In the conclusion of the rule we move into context $\Sigma, t : \exists R.C \centerdot$ where $t$ represents an $R$-filler of type $C$ accessible from the current focus. We extract the $R$-filler from $t$ by way of the

pattern $\texttt{let}\, R!a.y = t\, \texttt{in}\, e$ in which the filler object provided by $t$ is bound to $a$ and its method to $y$, respectively. Using $a$ and $y$ the expression $e$ then constructs the desired return value of type $D$. The fact that $e$ in the premise is constructed from the extended context $\Sigma\mathbf{.} \gg_{R?a} y : C$ ensures that $y$ is located at $a$ which is $R$-accessible from the scope of $\texttt{let}\, R!a.y = t\, \texttt{in}\, e$ which lives in context $\Sigma, t : \exists R.C\,\mathbf{.}$.

- The left rule $\forall L$ allows us to propagate context objects along sequences of role fillers all the way down from the root assumption to the current focus. Reading the rule upwards it says that whenever there is an object universally typed $t : \forall S.C$ in some local scope from where an $S$-filler with reference $b$ is directly reachable, then we can localise $t$ to $b$ and obtain an object method $t@b : C$ at $b$. Notice that $\forall L$ is applicable regardless the position of the focus.
- Finally, the right introduction rule $\forall R$ encodes input abstraction for $R$-filler locations: For an input action $R?a.e$ to have type $\forall R.D$ in context $\Sigma$ we consider a fresh $R$-filler $a$ for the current focus, extend the context to $\Sigma \gg_{R?a} \mathbf{.}\emptyset$ and type the body $e$ as $D$. The new context moves the focus forward into the scope of the $R$-filler $a$ where it starts with an empty scope to make sure that $e$ does not depend on any information other than what can be propagated from the old context $\Sigma$ across role $R$ to location $a$. Since $a$ is fresh and thus does not appear in $\Sigma$ we guarantee that $e$ works on any location $a$ as expressed by the universal quantifier $\forall R.D$.

The typing rules of Figs. 2 and 3 have the interesting feature that all terms which they type are in normal form, i.e., which are fully evaluated.We obtain non-normal $\lambda_{\mathcal{ALC}}$ terms by substitution, governed by the rules given in Fig. 4. There are two ways to substitute, associated with the two possible typing statements $\Sigma \vdash e : D$ and $\Sigma \vdash \emptyset \gg_{R!k} e : D$: The first rule $subst_1$ takes an expression $e$ typed in the current focus and substitutes it into another expression (statement) $\Psi$ for a free variable $x : D$ in the current scope. The second rule $subst_2$ permits us to take an expression $e : D$ typed in the scope of an $R$-filler $k$ relative to the current focus and substitute both into expression $\Psi$ which depends on $R$-filler reference $a$ and filler method $x$.

$$\frac{\Sigma\mathbf{.} \vdash e : D \qquad \Sigma, x : D\mathbf{.} \vdash \Psi}{\Sigma\mathbf{.} \vdash \Psi[e/x]}\; subst_1$$

$$\frac{\Sigma\mathbf{.} \vdash \emptyset \gg_{R!k} e : D \qquad \Sigma\mathbf{.} \gg_{R?a} x : D \vdash \Psi}{\Sigma\mathbf{.} \vdash \Psi[e/x, k/a]}\; subst_2$$

The conclusion statements $\Psi[e/x]$, $\Psi[e/x, k/a]$ arise from $\Psi$ by substituting $e$ for every free occurrence of $x$ and $k$ for $a$ in $\Psi$.

Figure 4: Substitution Rules.

The type system is conservative over $c\mathcal{ALC}$ in the same way as the pure simply typed $\lambda$-calculus is conservative over intuitionistic propositional logic. Let $D$ be a concept description without negation or falsity. Then $D$ is a theorem of $c\mathcal{ALC}$ iff there exists a $\lambda_{\mathcal{ALC}}$ method that implements it, i.e., there is $m$ such that $\mathbf{.}\emptyset \vdash m : D$ is derivable. This follows from the completeness proof of the term-free calculus for $c\mathcal{ALC}$ [14].

The $c\mathcal{AL}$culus is pure in the sense that it does not prejudice the properties of roles and fillers. In particular, it does not assume that there are at all any fillers for a given role or that there are non-trivial methods which could be instantiated for some $R$-fillers. All information about roles and fillers must be axiomatised in the typing context. In fact, $\lambda_{\mathcal{ALC}}$ does not contain closed methods of type $\exists R.D$ for any $D$ and any closed method of type $\forall R.D$ is essentially of the form $R?a.m$ where $m$ has type $D$ and thus can be turned into a method $S?a.m : \forall S.D$ for all roles $S$.

**Example 3** (IK1, IK2)**.** The $K$-combinators of the Hilbert system for $c\mathcal{ALC}$ [14] corresponding to the axioms IK1, IK2 of CK (see p.54), respectively are the typed terms

$$\begin{array}{lcll} \mathsf{K}_{\exists R} & = & \lambda x.\,\lambda z.\,\texttt{let}\ R!a.y = z\ \texttt{in}\ R!a.((x@a)\,y) & : \forall R.(C \sqsubseteq D) \sqsubseteq (\exists R.C) \sqsubseteq (\exists R.D) \\ \mathsf{K}_{\forall R} & = & \lambda x.\,\lambda z.\,R?a.(x@a)(z@a) & : \forall R.\,(C \sqsubseteq D) \sqsubseteq (\forall R.C) \sqsubseteq (\forall R.D) \end{array}$$

obtained from the following derivations (without terms, for conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \gg_R .C \vdash C}\,Ax_m \quad \dfrac{}{\emptyset \gg_R .D, C \vdash D}\,Ax_m}{\emptyset \gg_R .C \sqsubseteq D, C \vdash D}\,\sqsubseteq L}{\forall R.\,(C \sqsubseteq D) \gg_R .C \vdash D}\,\forall L}{.\forall R.\,(C \sqsubseteq D) \gg_R C \vdash \emptyset \gg_R D}\,Ax_f}{.\forall R.\,(C \sqsubseteq D) \gg_R C \vdash \exists R.D}\,\exists R}{.\forall R.\,(C \sqsubseteq D), \exists R.C \vdash \exists R.D}\,\exists L}{.\forall R.\,(C \sqsubseteq D) \vdash (\exists R.C) \sqsubseteq (\exists R.D)}\,\sqsubseteq R}{.\emptyset \vdash \forall R.\,(C \sqsubseteq D) \sqsubseteq (\exists R.C) \sqsubseteq (\exists R.D)}\,\sqsubseteq R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \gg_{R?a} .C \vdash C}\,Ax_m \quad \dfrac{}{\emptyset \gg_{R?a} .D, C \vdash D}\,Ax_m}{\emptyset \gg_{R?a} .C \sqsubseteq D, C \vdash D}\,\sqsubseteq L}{\forall R.\,(C \sqsubseteq D) \gg_{R?a} .C \vdash D}\,\forall L}{\forall R.\,(C \sqsubseteq D), \forall R.C \gg_{R?a} .\emptyset \vdash D}\,\forall L}{.\forall R.\,(C \sqsubseteq D), \forall R.C \vdash \forall R.D}\,\forall R}{.\forall R.\,(C \sqsubseteq D) \vdash (\forall R.C) \sqsubseteq (\forall R.D)}\,\sqsubseteq R}{.\emptyset \vdash \forall R.\,(C \sqsubseteq D) \sqsubseteq (\forall R.C) \sqsubseteq (\forall R.D)}\,\sqsubseteq R$$

□

**Example 4** (MP and Nec)**.** Our calculus maintains the crucial distinction between the rules of Modus Ponens and Necessitation which is that the former can be internalised as a closed term (known as *functional completeness*) but the latter cannot. Modus Ponens, which is simply function application, is representable as a combinator $\mathsf{MP} = \lambda y.\,\lambda x.\,y\,x$ for which $.\emptyset \vdash \mathsf{MP} : C \sqsubseteq (C \sqsubseteq D) \sqsubseteq D$ is derivable. Then, if $.\emptyset \vdash m_1 : C \sqsubseteq D$ and $.\emptyset \vdash m_2 : C$ we have $.\emptyset \vdash \mathsf{MP}\,m_1\,m_2 : D$. What about Necessitation? Is there also a combinator $.\emptyset \vdash \mathsf{Nec}_R : D \sqsubseteq \forall R.D$ such that $.\emptyset \vdash m : D$ gives $.\emptyset \vdash \mathsf{Nec}_R\,m : \forall R.D$? Indeed, context weakening guarantees that $.\emptyset \vdash m : D$ implies $.\emptyset \vdash R?a.m : \forall R.D$ for any role $R$. Hence, $\mathsf{Nec}_R\,m$ is an abbreviation for $R?a.m$. But this does not mean that $\mathsf{Nec}_R$ is representable as a (first-order) operator $\lambda x.\,R?a.\,x$. In fact, the attempt to type this term

$$\dfrac{\dfrac{\dfrac{}{x : D \gg_{R?a} .\emptyset \vdash x : D}\,???}{.x : D \vdash R?a.\,x : \forall R.D}\,\forall R}{.\emptyset \vdash \lambda x.\,R?a.\,x : D \sqsubseteq \forall R.D}\,\sqsubseteq R$$

fails because the axiom rule $Ax_m$ does not apply for sequent $x : D \gg_{R?a} .\emptyset \vdash x : D$ in which variable $x$ appears in the context outside of the current focus. In order to bring it into focus we need rule $\forall L$. However, for $\forall L$ to be applicable, we should have type $x : \forall R.D$ for the variable. The resulting derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \gg_{R?a} .x@a : D \vdash x@a : D}\,Ax_m}{x : \forall R.D \gg_{R?a} .\emptyset \vdash x@a : D}\,\forall L}{.x : \forall R.D \vdash R?a.\,x@a : \forall R.D}\,\forall R}{.\emptyset \vdash \lambda x.\,R?a.\,x@a : \forall R.D \sqsubseteq \forall R.D}\,\sqsubseteq R$$

yields the identity function $\lambda x.\,R?a.\,x@a \cong \lambda x.\,x$[1] which is fine but something entirely different from $\mathsf{Nec}_R$. □

[1] This is a form of $\eta$-conversion $R?a.\,x@a \cong x$ just like $\lambda y.\,f\,y \cong f$ in the $\lambda$-calculus.

**Example 5** (Disjunctive Distribution)**.** In [15] it was pointed out as one of the constructive features of $c\mathcal{ALC}$ that $\exists R$ does not distribute over disjunction, which contrasts with classical $\mathcal{ALC}$. This was motivated by model-theoretic means. We can now justify operationally why the concept description $\exists R.(C \sqcup D) \sqsubseteq (\exists R.C \sqcup \exists R.D)$ is not typeable in $\lambda_{\mathcal{ALC}}$. Exploiting cut elimination (normal form terms) the only possible candidate method $m$ which would generate $m\,x : \exists R.C \sqcup \exists R.D$ from $x : \exists R.(C \sqcup D)$ is

$$m \quad =_{df} \quad \lambda x.\ \mathtt{let}\ R!a.y = x\ \mathtt{in}\ \mathtt{case}\ y\ \mathtt{of}[\iota_1\, y_1 \to \iota_1\, R!a.y_1 \mid \iota_2\, y_2 \to \iota_2\, R!a.y_2] \tag{6}$$

but its typing

$$\dfrac{\dfrac{\dfrac{}{\centerdot\emptyset \gg_{R?a}\ y : C \sqcup D \vdash \mathtt{case}(a,y) : \exists R.C \sqcup \exists R.D}\ ??}{\centerdot x : \exists R.(C \sqcup D) \vdash \mathtt{let}\ R!a.y = x\ \mathtt{in}\ \mathtt{case}(a,y) : \exists R.C \sqcup \exists R.D}\ \exists L}{\centerdot\emptyset \vdash \lambda x.\,\mathtt{let}\ R!a.y = x\ \mathtt{in}\ \mathtt{case}(a,y) : \exists R.(C \sqcup D) \sqsubseteq \exists R.C \sqcup \exists R.D}\ \sqsubseteq R$$

where $\mathtt{case}(a,y)$ abbreviates $\mathtt{case}\ y\ \mathtt{of}[\iota_1\, y_1 \to \iota_1\, R!a.y_1 \mid \iota_2\, y_2 \to \iota_2\, R!a.y_2]$ does not quite work out. At the top line the only rule applicable is $\sqcup R$ which is certainly not sensible as it would break up the statement (output) disjunction $\exists R.C \sqcup \exists R.D$ on the output side before we have split the two cases of $y : C \sqcup D$ in the context (input). Yet, rule $\sqcup L$ for case analysis on $y$ does not apply because it does not have the focus. Trouble is, there is no way to move the focus right to get

$$\emptyset \gg_{R?a} \centerdot y : C \sqcup D \vdash \mathtt{case}(a,y) : \exists R.C \sqcup \exists R.D$$

since the rule $Ax_f$ which would do that is not applicable at this point. So, what is the problem? Look at the typing that we need:

$$\centerdot\emptyset \gg_{R?a} y : C \sqcup D \vdash \mathtt{case}(a,y) : \exists R.C \sqcup \exists R.D$$

It forces us to make a decision between $\exists R.C$ or $\exists R.D$ in the scope which has focus and this is empty. Instead, we need to access the method $y : C \sqcup D$ in the scope of filler $a$ but this is one $R$-role ahead from the object in scope.

Suppose we evaluated our candidate term $m\,x$ from (6) in the scope of some entity $b$. The pattern matching $\mathtt{let}\ R!a.y = x$ extracts an $R$-filler of $b$ from $x$, binds it to $a$ and also a method of type $C \sqcup D$ bound to variable $y$. Then, a case analysis on $y$ is made: if $y$ is a left injection $\iota_1\, y_1$ we return $\iota_1\, R!a.y_1$. Otherwise, if $y$ is a right injection we take $\iota_2\, R!a.y_2$ as result. However, both these computations are done at filler $b$ whereas the value $y$ on which they depend lives at $R$-filler $a$. Our pure type system precludes such scope extrusions. The pure $c\mathcal{AL}$culus is conservative and does not permit forward references to use computations from an $R$-filler $a$ to construct values at its predecessor $b$. This allows for more general classes of interpretations. Some interpretation may permit forward references, some only in special situations, others never. This depends on applications and the choice of a suitable context theory (computational TBox). □

**Example 6** (IK5)**.** Another axiom that is not valid in $c\mathcal{ALC}$ is IK5. This has been motivated model-theoretically in [14]. At this point we can give an operational justification why the concept description $(\exists R.C \sqsubseteq \forall R.D) \sqsubseteq \forall R.(C \sqsubseteq D)$ is not typeable in $\lambda_{\mathcal{ALC}}$. One possible expression $m$ with the type of IK5 is

$$m \quad =_{df} \quad \lambda x.\, R?a.(\lambda y.\, ((x\ (R!a.y))@a)). \tag{7}$$

but the type check for $m$ fails as can be seen in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \exists R.C \sqsubseteq \forall R.D \;\gg_{R?a}\; \centerdot\, y : C \vdash (x(R!a.y))@a : D}\,??}{x : \exists R.C \sqsubseteq \forall R.D \;\gg_{R?a}\; \centerdot \emptyset \vdash \lambda y.\,((x(R!a.y))@a) : C \sqsubseteq D}\,\sqsubseteq R}{\centerdot x : \exists R.C \sqsubseteq \forall R.D \vdash R?a.(\lambda y.\,((x(R!a.y))@a)) : \forall R.(C \sqsubseteq D)}\,\forall R}{\centerdot \emptyset \vdash \lambda x.\,R?a.(\lambda y.\,((x(R!a.y))@a)) : (\exists R.C \sqsubseteq \forall R.D) \sqsubseteq \forall R.(C \sqsubseteq D)}\,\sqsubseteq R$$

At the top line no more rule is applicable. The problem lies in the context localities of $x$ and $y$, i.e. in the application $(x\ (R!a.y))$ where $x$ is from the outer context, the term $(R!a.y)$ depends on $y$ from the inner context and $y$ lies behind one $R$-step. Since $y$ is $R$-context dependent w.r.t. $a$, it cannot be used outside of this context. In a system where context-awareness plays a role it is not possible for a context dependent term (context locally) to be used in a global fashion (context globally). The focus constrains the derivations to context-aware sequents. Formally, this can be demonstrated by allowing the focus to move arbitrarily backward and forward. Moving the focus in the top sequent to the left gives us the typing we need

$$\centerdot x : \exists R.C \sqsubseteq \forall R.D \;\gg_{R?a}\; y : C \vdash (x(R!a.y))@a : D$$

such that we can proceed with rule $\sqsubseteq L$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \;\gg_{R?a}\; \centerdot\, y : C \vdash y : C}\,Ax_m}{\emptyset\centerdot \;\gg_{R?a}\; y : C \vdash \emptyset \gg_{R?a}\; y : C}\,Ax_f}{\emptyset\centerdot \;\gg_{R?a}\; y : C \vdash R!a.y : \exists R.C}\,\exists R \qquad \dfrac{\dfrac{}{\emptyset\centerdot \;\gg_{R?a}\; (x\ (R!a.y))@a : D, y : C \vdash (x\ (R!a.y))@a : D}\,??}{x\ (R!a.y) : \forall R.D\centerdot \;\gg_{R?a}\; y : C \vdash (x\ (R!a.y))@a : D}\,\forall L}{\centerdot x : \exists R.C \sqsubseteq \forall R.D \;\gg_{R?a}\; y : C \vdash (x(R!a.y))@a : D}\,\sqsubseteq L$$

This allows us to use the term $R!a.y$ outside of the $R$-context in the left branch of the derivation and permits the construction of the application term $(x\ (R!a.y))$ of type $\forall R.D$. From the current focus this term can be propagated by rule $\forall L$ along the existing $R$-filler $a$, i.e. localised at $a$, to obtain the object method $(x\ (R!a.y))@a : D$ at $a$. But the application of Rule $Ax_m$ and therefore the typing is not possible since the focus is at the outer context. To close the right branch of the derivation we have to move the focus again, this time into the $R$-context which gives us the sequent

$$\emptyset \;\gg_{R?a} \centerdot\, (x\ (R!a.y))@a : D, y : C \vdash (x\ (R!a.y))@a : D$$

that can be closed by rule $Ax_m$. Our pure system excludes such arbitrary context changes and does not allow a context dependent term to be used independently from its context. □

# 4 Operational Semantics

The computational semantics of $\lambda_{\mathcal{ALC}}$ is given in the standard fashion by term rewriting, starting with the basic contractions of Fig. 5. Of those, the contractions $\beta\pi_1$, $\beta\pi_2$, $\beta\mathtt{case}_1$, $\beta\mathtt{case}_2$, $\beta\lambda$ are well known from $\lambda$-calculus [9] while $\beta R?$, $\beta R!$ are new for $\lambda_{\mathcal{ALC}}$. We write $e \twoheadrightarrow e'$ for the reflexive, transitive closure of contractions applied in arbitrary sub-expressions of $e$.

An input prefix $R?a.\,e$ acts like a functional abstraction $\lambda a.\,e$ where the role $R$ indicates a named access method or named channel unlike the $\lambda$ binder which is anonymous. Accordingly,

$$
\begin{array}{rcll}
\pi_1(e_1, e_2) & \longrightarrow & e_1 & \beta\pi_1 \\
\pi_2(e_1, e_2) & \longrightarrow & e_2 & \beta\pi_2 \\
\texttt{case}\ \iota_1\, e\ \texttt{of}\ [\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2] & \longrightarrow & e_1\{e/x_1\} & \beta\texttt{case}_1 \\
\texttt{case}\ \iota_2\, e\ \texttt{of}\ [\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2] & \longrightarrow & e_2\{e/x_2\} & \beta\texttt{case}_2 \\
(\lambda x.e_1)\, e_2 & \longrightarrow & e_1\{e_2/x\} & \beta\lambda \\
(R?a.\, e)@b & \longrightarrow & e\{b/a\} & \beta R? \\
\texttt{let}\ R!a.x = R!k.e_1\ \texttt{in}\ e_2 & \longrightarrow & e_2\{e_1/x\}\{k/a\} & \beta R!
\end{array}
$$

Figure 5: Contraction Rules ($\beta$-Reductions).

filler instantiation $e@a$ corresponds to function application with contraction $\beta R?$ mimicking the $\beta$-reduction for $\lambda$. In contrast to ordinary $\lambda$-terms the prefix $R?a.\, e$ only abstracts filler object references $a$ rather than the filler methods. The filler references themselves are not typed, only their methods are. The expression $\texttt{let}\ R!a.y = e\ \texttt{in}\ e_2$ is a pattern matching which opens an $R$-filler $e$ binding its object reference to variable $a$ and object methods to variable $y$ and executes expression $e_2$. Hence, if $e$ has been evaluated to explicit form $R!k.e_1$ the bindings in $e_2$ can be executed with $e_2\{e_1/x\}\{k/a\}$ as done in contraction $\beta R!$.

The operational semantics exhibit a perfect duality in the syntax for role filling: $R?$ and $R!$ are the *constructors* for types $\forall R.C$ and $\exists R.C$, respectively, while @ and $\texttt{let}$ are the associated *destructors*. The contraction rules implement the idea that applying a destructor on top of a constructor returns the original parts from which the object was constructed. Moreover, the difference between $R?$ and $R!$ is the difference between input and output. To see this let us consider reductions $p \twoheadrightarrow R?a.q$ or $p \twoheadrightarrow R!k.q$ for a closed expression $p : \forall R.C$ or $p : \exists R.C$, respectively, as abstract *role filling transitions*

$$p \overset{R?a}{\twoheadrightarrow} q \qquad p \overset{R!k}{\twoheadrightarrow} q.$$

which move from an object $p$ to a filler object $q$ through input and output *actions* $R?a$, $R!k$. Now look at the computational behaviour of the two combinators $\mathsf{K}_{\exists R}$, $\mathsf{K}_{\forall R}$ from Ex. 3 which encapsulate two forms of function application under role filling. Based on the reduction rules from Fig. 5 we obtain the following evaluation rules

$$\frac{f \overset{R?b}{\twoheadrightarrow} f' \qquad p \overset{R!k}{\twoheadrightarrow} p'}{\mathsf{K}_{\exists R}\, f\, p \overset{R!k}{\twoheadrightarrow} (f'\{k/b\})\, p'} \qquad \frac{f \overset{R?b}{\twoheadrightarrow} f' \qquad p \overset{R?c}{\twoheadrightarrow} p'}{\mathsf{K}_{\forall R}\, f\, p \overset{R?a}{\twoheadrightarrow} (f'\{a/b\})(p'\{a/c\})}$$

In both cases, a function $f'$ is applied to an argument $p'$ which are obtained by role filling from $f$ and $p$, respectively. However, the filler scope at which $f'\, p'$ is executed (or available) is controlled differently. In $\mathsf{K}_{\exists R}\, f\, p$ the scope is determined by the argument $p$ and passed to function $f$ as well as to the environment using output action $R!k$. In contrast, the process $\mathsf{K}_{\forall R}\, f\, p$ takes the scope as an external input $R?a$ from the environment and passes it to $f$ and $p$, which of course must be able to accept it. Accordingly, $p$ has type $\forall R.C$ in the expression $\mathsf{K}_{\forall R}\, f\, p$ while it is of type $\exists R.C$ in $\mathsf{K}_{\exists R}\, f\, p$. The function $f$ in both cases receives the filler scope in line with its universal type $f : \forall R.(C \sqsubseteq D)$ and thus is executable at any filler reference.

The contraction rules for the two variable binding operations $\texttt{case}\, e\, \texttt{of}$ and $\texttt{let}\ R!a.y = e\ \texttt{in}$ depend on their main sub-expression $e$ to be normalised to a constructor first, before they can fire. When dealing with open terms, however, we cannot assume that $e$ always reduces to a constructor pattern. So, while the reduction rules in Fig. 5 are complete for closed expressions,

we need to add further *commuting conversions*. In general, we need to be able to push every destructor of $\lambda_{\mathcal{ALC}}$ through an occurrence of both $\mathtt{case}\, e\, \mathtt{of}$ and $\mathtt{let}\, R!a.y = e\, \mathtt{in}$. We only give the relevant cases for `let` in Fig. 6, the ones for `case` are analogous (cf. [9]).

$$
\begin{array}{lcll}
\pi_i(\mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, e_2) & \longrightarrow & \mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, \pi_i\, e_2 & \gamma R!_1 \\
(\mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, e_2)\, e_3 & \longrightarrow & \mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, e_2\, e_3 & \gamma R!_2 \\
(\mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, e_2)@k & \longrightarrow & \mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, e_2@k & \gamma R!_3 \\
\mathtt{case}(\mathtt{let}\, R!a.y = e_1\, \mathtt{in}\, e_2) & & & \\
\quad \mathtt{of}[\iota_1\, x_1 \to e_3 \mid \iota_2\, x_2 \to e_4] & \longrightarrow & \mathtt{let}\, R!a.y = e_1 & \\
 & & \quad \mathtt{in}\, \mathtt{case}\, e_2\, \mathtt{of}[\iota_1\, x_1 \to e_3 \mid \iota_2\, x_2 \to e_4] & \gamma R!_4 \\
\mathtt{let}\, R!a.y = \mathtt{let}\, S!b.z = e_1\, \mathtt{in}\, e_2\, \mathtt{in}\, e_3 & \longrightarrow & \mathtt{let}\, S!b.z = e_1\, \mathtt{in}\, \mathtt{let}\, R!a.y = e_2\, \mathtt{in}\, e_3 & \gamma R!_5 \\
\pi_i(\mathtt{case}\, x\, \mathtt{of}[\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2]) & \longrightarrow & \mathtt{case}\, x\, \mathtt{of}[\iota_1\, x_1 \to \pi_i\, e_1 \mid \iota_2\, x_2 \to \pi_i\, e_2] & \gamma\, \mathtt{case}_1 \\
(\mathtt{case}\, x\, \mathtt{of}[\iota_1\, x_1 \to e_1, \iota_2\, x_2 \to e_2])\, e_3 & \longrightarrow & \mathtt{case}\, x\, \mathtt{of}[\iota_1\, x_1 \to e_1\, e_3 \mid \iota_2\, x_2 \to e_2\, e_3] & \gamma\, \mathtt{case}_2 \\
(\mathtt{case}\, x\, \mathtt{of}[\iota_1\, x_1 \to e_1 \mid \iota_2\, x_2 \to e_2])@k & \longrightarrow & \mathtt{case}\, x\, \mathtt{of}[\iota_1\, x_1 \to e_1@k \mid \iota_2\, x_2 \to e_2@k] & \gamma\, \mathtt{case}_3 \\
\mathtt{case}(\mathtt{case}\, y\, \mathtt{of}[\iota_1\, y_1 \to e_1 \mid \iota_2\, y_2 \to e_2]) & & & \\
\quad \mathtt{of}[\iota_1\, x_1 \to e_3 \mid \iota_2\, x_2 \to e_4] & \longrightarrow & \mathtt{case}\, y\, \mathtt{of} & \\
 & & \quad [\iota_1\, y_1 \to \mathtt{case}\, e_1\, \mathtt{of}[\iota_1\, x_1 \to e_3 \mid \iota_2\, x_2 \to e_4], & \\
 & & \quad\ \iota_2\, y_2 \to \mathtt{case}\, e_2\, \mathtt{of}[\iota_1\, x_1 \to e_3 \mid \iota_2\, x_2 \to e_4]] & \gamma\, \mathtt{case}_4 \\
\mathtt{let}\, R!a.y = & & & \\
\quad \mathtt{case}\, e_1\, \mathtt{of}[\iota_1\, x_1 \to e_2 \mid \iota_2\, x_2 \to e_3]\, \mathtt{in}\, e_4 & \longrightarrow & \mathtt{case}\, e_1\, \mathtt{of} & \\
 & & \quad [\iota_1\, x_1 \to \mathtt{let}\, R!a.y = e_2\, \mathtt{in}\, e_4 \mid & \\
 & & \quad\ \iota_2\, x_2 \to \mathtt{let}\, R!a.y = e_3\, \mathtt{in}\, e_4] & \gamma\, \mathtt{case}_5
\end{array}
$$

Figure 6: Commuting Conversions.

Our language $c\mathcal{ALC}$ now consists of Figs. 2, 3 and 4 for typing and Figs. 5 and 6 for execution. As mentioned before, the subsystem of Figs. 2, 3 types precisely the normal forms. If $\Sigma \vdash e : D$ is derivable using the rules of Figs. 2 and 3 then $e \twoheadrightarrow e'$ implies $e = e'$. $\lambda_{\mathcal{ALC}}$ is a conservative extension of the simply typed $\lambda$-calculus which enjoys the same computational soundness properties: We conjecture that the reduction relation $\twoheadrightarrow$ is confluent and strongly normalising on well-typed expressions. It is a open problem whether the reduction relation $\twoheadrightarrow$ satisfies subject reduction i.e., $\Sigma \vdash e : D$ and $e \twoheadrightarrow e'$ implies $\Sigma \vdash e' : D$.

Thus, $\lambda_{\mathcal{ALC}}$ sets the ground for a higher-order functional framework for computing in DL-knowledge bases and DL data structures. It realises the Proposition-as-Types and Proofs-as-Programs Principle (Curry-Howard Isomorphism) for the constructive refinement $c\mathcal{AL}$ of classical $\mathcal{AL}$. Considering the closed normal form terms as the canonical values of a data type $D$, then the values of concept description $C \sqcup D$ are the disjoint union of the values of $C$ and $D$. Indeed, every closed method $m : C \sqsubseteq D$ normalises to $m \twoheadrightarrow \iota_1 m'$ where $m' : C$ or to $m \twoheadrightarrow \iota_2 m'$ and $m' : D$. Similarly, the values of $m : C \sqcap D$ are pairs $m = (m_1, m_2)$ of values of $C$ and $D$, respectively; the values of $m : C \sqsubseteq D$ are functions $m = \lambda x.e$ from values of $C$ to those of $D$; finally, every value of type $\exists R.C$ is an output action $m = R!k.m'$ with a filler reference $k$ and filler value $m' : D$ and every value of $\forall R.C$ is an input action $R?a.e$ which accepts an $R$-filler reference $k$ and returns filler methods $e\{k/a\}$.

# 5 Conclusion and Future Work

In this paper we present how the constructive description logic $c\mathcal{ALC}$ can serve as a semantic typing system for an extension of the simply typed lambda-calculus which is able to express

context-dependent computations in structured data, e.g. computational knowledge bases or databases. The typing system is derived from a multi-sequent calculus for $c\mathcal{ALC}$ which has been shown to be sound and complete as has been reported in [16].

As for future work, we aim to prove that $\lambda_{\mathcal{ALC}}$ exhibits the same computational soundness properties, namely strong normalisation, confluence and subject reduction, possibly restricted to special context aware rewriting strategies. The work presented here is still tentative and leaves many open questions.

We hope that such an approach can constitute a formal grounding for a modally typed functional programming language that finds practical adoption in the domain of knowledge base and database processing languages.

# References


[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementations and Applications*. Cambridge University Press, 2nd edition, 2007.

[3] G. Bellin, V. de Paiva, and E. Ritter. Extended Curry-Howard correspondence for a basic constructive modal logic. In *Methods for Modalities II*, November 2001.

[4] L. Botazzo, M. Ferrari, C. Fiorentini, and G. Fiorino. A constructive semantics for ALC. In *Int'l Workshop on Description Logics (DL 2007)*, pages 219–226, 2007.

[5] V. de Paiva. Constructive description logics: what, why and how. In *Context Representation and Reasoning*, Riva del Garda, August 2006.

[6] M. Fairtlough, M. Mendler, and M. Walton. First-order lax logic as a framework for constraint logic programming. Technical Report MIP-9714, University of Passau, July 1997.

[7] G. Fischer-Servi. Semantics for a class of intuitionistic modal calculi. In M. L. Dalla Chiara, editor, *Italian Studies in the Philosophy of Science*, pages 59–72. Reidel, 1980.

[8] D. M. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-dimensional modal logics*. Elsevier, 2003.

[9] J. Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[10] S. Kobayashi. Monad as modality. *Theoretical Computer Science*, 175:29–74, 1997.

[11] G. Luettgen and M. Mendler. The intuitionism behind Statecharts step. *ACM Transactions on Computational Logic*, 3(1):1–41, January 2002.

[12] M. Mendler. *A Modal Logic for Handling Behavioural Constraints in Formal Hardware Verification*. PhD thesis, Department of Computer Science, University of Edinburgh, ECS-LFCS-93-255, March 1993.

[13] M. Mendler and V. de Paiva. Constructive CK for contexts. In L. Serafini and P. Bouquet, editors, *Context Representation and Reasoning (CRR-2005)*, volume 13 of *CEUR Proceedings*, July 2005.

[14] M. Mendler and S. Scheele. Cut-free gentzen calculus for multimodal $\mathsf{CK}^*$. *To appear in special issue of Information and Computation on Intuitionistic Modal Logics and Applications (IMLA)*.

[15] M. Mendler and S. Scheele. Towards constructive description logics for abstraction and refinement. In *21st Int'l Workshop on Description Logics (DL2008)*. CEUR Workshop proceedings 353, May 2008.

[16] M. Mendler and S. Scheele. Towards constructive dl for abstraction and refinement. *J. Autom. Reason.*, 44(3):207–243, 2010.

[17] R. Möller. A functional layer for description logics: knowledge representation meets object-oriented programming. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-*

*oriented programming, systems, languages, and applications*, pages 198–213, New York, NY, USA, 1996. ACM.

[18] A. Nanevski. From dynamic binding to state via modal possibility. In *Int'l. Conf. on PRinciples and Practice of Declarative Programming (PPDP'03)*, pages 207–218, Uppsala, Sweden, 2003.

[19] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, V(N):1–48, March 2007.

[20] A. Paschke. Typed hybrid description logic programs with order-sorted semantic web type systems on OWL and RDFS. Technical report, TU Munich, December 2005.

[21] F. Pfenning and R. Davies. A judgemental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, August 2001.

[22] G. Plotkin and C. Stirling. A framework for intuitionistic modal logics. In J.Y. Halpern, editor, *Theoretical aspects of reasoning about knowledge*, Monterey, 1986.

[23] S. Schacht and U. Hahn. A denotational semantics for joining description logics and object-oriented programming. In *SCAI '97: Proceedings of the sixth Scandinavian conference on Artificial intelligence*, pages 119–130, Amsterdam, The Netherlands, The Netherlands, 1997. IOS Press.

[24] A.K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[25] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics*, volume II. North-Holland, 1988.

[26] D. van Dalen. Intuitionistic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume III, chapter 4, pages 225–339. Reidel, 1986.

[27] D. Wijesekera. Constructive modal logic I. *Annals of Pure and Applied Logic*, 50:271–301, 1990.