The syntax of many-valued relations

P. Eklund[1]

Umeå University, Department of Computing Science, Umeå, Sweden

Abstract. In this paper we show how many-valued relations syntactically can be formulated using powertype constructors. This in turn enables to describe the syntax of generalized relations in the starting point sense where the category sets and relations is isomorphic to the Kleisli category of the powerset monad over the category of sets. We can then generalize to work over monoidal closed categories, and thereby description logic, formal concepts and rough sets can be viewed as depending on that powertype constructor, and within a setting of many-valued λ-calculus. In order to achieve this, we will adopt a three-level arrangement of signatures [3], and demonstrate the benefits of using it.
Keywords: Concept, functor, generalized relation, monad, signature.

1 Introduction

Terms are the foundational cornerstones of logic, and signatures are foundational for the term construction. Type constructors are frequently used to create new sorts from old ones, but type constructors are traditionally adopted *from the outside*. This means that type constructors are not seen as operators in a signature in its own right.

We adopt a three-level arrangement of signatures [3] where the middle level contains type constructors, and the first and third level, respectively, is used to clearly distinguish terms from λ-terms. The conventional definition of λ-terms is informal, and, in fact, not constructive, or at least to say that it hides the underlying formal term construction. Doing so, the conventional definition creates a demand for renaming, which cannot be formally justified, but is an *ad hoc* necessity to avoid ambiguities. Similar hidden phenomena appears, of course, in many branches of computing, even in Turing machines, that hides recursion in a way that makes Church's thesis to be informal only.

We respect Church's view that λ is just an *informal symbol* [2], and we go even further by showing how it can be formalized, when it must be formalized. A fundamental consequence for terms on *signature levels* is then that λ is seen not to be a *general abstractor*, but rather that any operator possesses its own capacity to abstract itself, and not to possess any capacity whatsoever to abstract anything else. In an expression like $\lambda x.f$, λ is unique to f, and should be clearly viewed as "f owns its λ". This obviously departs from traditional views and definitions of λ-terms, but on the other hand it disables the appearance of "unwanted" terms. Doing so we, we can also avoid renaming. In fact, our approach on avoiding renaming goes far beyond de Bruijn's notation [1].

A key enabler is then the possibility to use a wide variety of further type constructor, together with the function type constructor producing λ-calculus. The type constructors enable the *transportation* of terms on level one in the three-level arrangement of signatures to become abstracted to λ-terms on the third level. A key issue for the *syntax of relations*, and e.g. for fuzzy description logic, formal concepts and rough sets, is the use of the powertype constructor. This also enables to describe the syntax of generalized relations in the sense where the category of sets and relations is isomorphic to the Kleisli category of the powerset monad over the category of sets.

For many-valuedness to become identifiable within and because of the use of suitable underlying categories, we use monoidal categories, and thereby the notion of many-valued description logic, formal concepts [5] and rough sets [4], can be seen as depending on that powertype constructor. In this paper we focus on description logic.

2 Informally defined terms in type theory

The informal definition of (untyped) λ-terms basically states, firstly, that a variable is a λ-term, secondly, if M is a λ-term, then $\lambda x.M$ is a λ-term, where x is a variable (abstraction), and thirdly, if M and N are λ-terms, then also MN is a λ-term (application).

Traditional λ-calculus looks at terms as becoming abstracted to operators. It is generally seen as a nice trick, but cannot be formally and logically justified. What in fact happens is that an operator is abstracted to another operator, of different arity. Variables cause confusion, and the notion of *free* and *bound* is the root of this inconvenience, trying to make substitution something more than it actually is. Substitution is a morphism in the Kleisli category of a given term monad and over a selected category [3]. Church [2] indeed called "λ" an *improper symbol*, together with "(" and ")" also being improper symbols. The *proper symbols* are those residing in the signature, or being symbols for variables. Church also states the following: *A complete incorporation of the calculus of λ-conversion into the theory of types is impossible if we require that λ and juxtaposition shall retain their respective meanings as an abstraction operator and as denoting the application of function to argument.* This obviously means that λ is not to be seen as an operator itself appearing in some signature.

Note also that Church's ι and o types are not clearly defined. There is a consensus about ι being the 'type of types', but we have to be careful e.g. not say that "ι is a ι". This creates problems, and modern type theory still struggles with this issue. The o type for 'propositions' is also still not explained in a satisfactory way. Church did say that *o is the type of propositions*, but he also states the following: *We purposely refrain from making more definite the nature of the types o and ι, the formal theory admitting of a variety of interpretations in this regard. Of course the matter of interpretation is in any case irrelevant to the abstract construction of the theory, and indeed other and quite different interpretations are possible (formal consistency assumed).*

3 Levels of signatures

The syntax of relations, i.e., the powertype, resides as a unary operator on the second level in a three-level arrangement of signatures. Generalized relations, as syntactic objects are therefore λ-terms in a general sense, i.e., generality depending on the semantics of the operator, but in particular, as we shall see, on the underlying category of the term monad producing *types as terms* on that second level for the sake of delivering generalized λ-terms over the third level signature.

In order to explain this in all detail, we adopt the categorically somewhat informal notation $\Sigma = (S, \Omega)$ of a signature. The way it actually needs to be handled in a more strict fashion is explained in [3], which also contains detail on the corresponding formal and fully categorical construction of the term monad with its underlying term functor T_Σ. For a sort $\mathtt{s}$, and a term t of sort $\mathtt{s}$, we may use the notation $t :: \mathtt{s}$. The underlying category is some monoidal biclosed category, but in this treatment we hide detail about this underlying category.

On *level one*, we have Σ, and terms over Σ are produced by $\mathsf{T}_\Sigma X$, where X is an *object of variables* in the underlying category. In case of a one-sorted signature over the category of sets and functions, terms are just traditional terms as typically seen in first-order logic.

On *level two*, the level of type constructors, with introduce the single-sorted signature

$$\mathtt{S}_\Sigma = (\{\mathtt{type}\}, \{\mathtt{s} :\to \mathtt{type} \mid \mathtt{s} \in S\} \cup \{\Rrightarrow : \mathtt{type} \times \mathtt{type} \to \mathtt{type}\}),$$

where we then have $\mathsf{T}_{\mathtt{S}_\Sigma}\emptyset$ as the object of all types and constructed types.

On *level three*, the level then includes λ-terms based on the signature $\Sigma' = (S', \Omega')$ where

$$S' = \mathsf{T}_{\mathtt{S}_\Sigma}\emptyset$$

and Ω' is

$$\{\lambda^\omega_{i_1,\ldots,i_n} :\to (\mathtt{s}_{i_1} \Rrightarrow \cdots \Rrightarrow (\mathtt{s}_{i_{n-1}} \Rrightarrow (\mathtt{s}_{i_n} \Rrightarrow \mathtt{s})) \mid \omega : \mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s} \in \Omega\}$$

included with the operator

$$\mathtt{app}_{\mathtt{s},\mathtt{t}} : (\mathtt{s} \Rrightarrow \mathtt{t}) \times \mathtt{s} \to \mathtt{t}.$$

In this notation $(i_1, \ldots, i_n)$ is a permutation of $(1, \ldots, n)$. Note also how level one operators are transformed to constant operators on level three. Further, note indeed how "ω owns its abstraction" in $\lambda^\omega_{i_1,\ldots,i_n}$. In fact, we could even avoid using the informal symbol 'λ' in this context.

As an example, consider the signature of natural numbers

$$\mathtt{NAT} = (\{\mathtt{nat}\}, \{\mathtt{0} :\to \mathtt{nat}, \mathtt{succ} : \mathtt{nat} \to \mathtt{nat}\}$$

on level one. The 0-ary operator $\mathtt{0}$ converts to $\lambda^0_0 :\to \mathtt{nat}$, i.e., as a 0-ary operator on level three. The unary operator $\mathtt{succ}$ is (λ-)abstracted to become a 0-ary

operator $\lambda_1^{\mathtt{succ}} :\rightarrow (\mathtt{nat} \Rrightarrow \mathtt{nat})$ on level three. Note also that we must not confuse $\mathtt{nat}$ on level one with $\mathtt{nat}$ on level three, even if for simplicity we use the same notation.

We can now see the advantage in avoiding the need of renaming. In the traditional notation in λ-calculus, substituting x by $\mathtt{succ}(y)$ in

$$\lambda y.\mathtt{succ}(x)$$

requires a renaming of the bound variable y, e.g., $\lambda z.\mathtt{succ}(\mathtt{succ}(y))$. Such a renaming can now be avoided. On level one we have the substitution

$$\sigma_{\mathtt{nat}} : X_{\mathtt{nat}} \longrightarrow \mathsf{T}_{\mathtt{NAT},\mathtt{nat}}(X_{\mathtt{t}})_{\mathtt{t}\in\{\mathtt{nat}\}}$$

where $\sigma_{\mathtt{nat}}(x) = \mathtt{succ}(y)$, and x being a variable on level one. The extension of

$$\sigma_{\mathtt{nat}}$$

is

$$\mu_{X_{\mathtt{nat}}} \circ \mathsf{T}_{\mathtt{NAT},\mathtt{nat}}\,(\sigma_{\mathtt{t}})_{\mathtt{t}\in\{\mathtt{nat}\}} : \mathsf{T}_{\mathtt{NAT},\mathtt{nat}}(X_{\mathtt{t}})_{\mathtt{t}\in\{\mathtt{nat}\}} \longrightarrow \mathsf{T}_{\mathtt{NAT},\mathtt{nat}}(X_{\mathtt{t}})_{\mathtt{t}\in\{\mathtt{nat}\}}.$$

On level three we then have the substitution

$$\sigma_{\mathtt{nat}'} : X_{\mathtt{nat}'} \longrightarrow \mathsf{T}_{\mathtt{NAT}',\mathtt{nat}'}(X_{\mathtt{t}})_{\mathtt{t}\in S'}$$

with $\sigma_{\mathtt{nat}'}(x) = \mathtt{app}_{\mathtt{nat}',\mathtt{nat}'}(\lambda_1^{\mathtt{succ}}, x)$, x being a variable on level three, so

$$\mu_{\mathtt{nat}'} \circ \mathsf{T}_{\mathtt{NAT}',\mathtt{nat}'}\,\sigma_{\mathtt{nat}'}(\mathtt{app}_{\mathtt{nat}',\mathtt{nat}'}(\lambda_1^{\mathtt{succ}}, x))$$

requires no renaming.

Many-valued, or fuzzy, λ-terms can be introduced either using monad compositions, or allowing $\mathsf{T}_{\Sigma'}$ to be a functor over an underlying category (Goguen) $\mathtt{Set}(\mathfrak{Q})$, where $\mathfrak{Q}$ typically is a quantale.

On β-reduction we obviously have the following transition from the traditional form to using the three-level signature. Let $[x := t]$ be a substitution, i.e., we have some $\sigma(x) = t$, and choose a $\omega : \mathtt{s}_1 \times \mathtt{s}_2 \longrightarrow \mathtt{s}$. Then β-reduction

$$\lambda x.\lambda y.\omega(x, y)\ t \rightarrow_\beta \lambda y.(\omega(x, y)[x := t]) = \lambda y.\omega(t, y) :: \mathtt{s}_2 \Rrightarrow \mathtt{s}$$

transforms to

$$(\mu \circ \mathsf{T}\sigma)(\mathtt{app}(\lambda^{\omega}_{\mathtt{s}_1,\mathtt{s}_2}, x)) \rightarrow_\beta \mathtt{app}(\lambda^{\omega}_{\mathtt{s}_1,\mathtt{s}_2}, t) :: \mathtt{s}_2 \Rrightarrow \mathtt{s}.$$

All these constructions can potentially be used in natural language expressions involving modifiers and quantifiers in expresions like "there are more small balls than large balls in this box". Obviously, there are no unique solutions to handle this as they are context dependent. Possible encodings of such expression, or related subexpressions, in our three level signatures setting for λ-terms, could view modifiers are closely related to type constructors. Modifiers as operators on level three are then specified using constructed types on level two. We should note that quantifiers are more like abstractors of sentences, and it may therefore be anticipated that the formalization of quantifiers, with quantifier symbols as informal symbols, is similar to the formalization of the way λ acts on expressions.

4 The syntax of generalized relations

The observation that relations $R \subseteq X \times X$ correspond precisely to functions (in form of substitutions) $\sigma_R : X \longrightarrow \mathsf{P}X$, where P is the powerset functor over the category of sets and functions, is the basis for viewing *generalized relations* as morphisms (substitutions) in the Kleisli category over *generalized powerset monads*.

For $\Sigma = (S, \Omega)$ on level one, we now extend $\mathtt{S}_\Sigma$ with further operators beyond just $\Rrightarrow$. Concerning unary operators we may include an $\mathtt{F} : \mathtt{type} \to \mathtt{type}$, which intuitively is expected to be semantically described by a functor, that is, assuming that the 'algebra' of $\mathtt{type}$ is the class of objects in some monoidal closed category. Whereas the algebras $\mathfrak{A}(\Sigma)$ of signatures Σ, involving assignments of sorts $\mathtt{s}$ to domains $\mathfrak{A}(\mathtt{s})$ of $\mathfrak{A}$, are standard according to universal algebra, the 'algebra' of the (Σ-)superseding type signature $\mathtt{S}_\Sigma$ is not immediate since the domain assigned to the sort $\mathtt{type}$ clearly cannot be just a set. There are several options for this, and these considerations may go beyond traditional universal algebra. These discussions are outside the scope of this paper.

The 'syntactic functors' view is based on unary type constructors $\phi, \psi : \mathtt{type} \to \mathtt{type}$, allowing the *composed type constructor* $\psi \circ \phi : \mathtt{type} \to \mathtt{type}$ by $(\psi \circ \phi)\mathtt{s} = \psi(\phi\mathtt{s})$. For unary type constructors $\phi, \psi : \mathtt{type} \to \mathtt{type}$, a *type transformation* τ *from* ϕ *to* ψ, denoted $\tau : \phi \Rightarrow \psi$, if it exists, is assumed, for all $\mathtt{s} \in \mathsf{T}_{\mathtt{S}_\Sigma}\emptyset$, to be given by a unique (constant operator) $\tau_\mathtt{s} :\to (\phi\mathtt{s} \Rrightarrow \psi\mathtt{s})$. Further, we assume that any $\mathtt{f} :\to (\mathtt{s} \Rrightarrow \mathtt{t})$ gives rise to a unique $\phi\mathtt{f} :\to (\phi\mathtt{s} \Rrightarrow \phi\mathtt{t})$.

Various 'syntactic set functors' can be introduced, including the 'powerset' type constructor $\mathsf{P} : \mathtt{type} \to \mathtt{type}$ on level two, intuitively thinking that the 'algebra' of P is the powerset functor, with the underlying monoidal closed category being the category of sets and functions.

5 Description logic

For transforming description logic into our categorical framework, we use notations in [6]. *Interpretations* $\mathcal{I} = (D^\mathcal{I}, \cdot^\mathcal{I})$, where $\cdot^\mathcal{I}$ *maps every concept description* to a subset of $D^\mathcal{I}$, use D for that universe, which should not be confused with D as used for concept descriptions, e.g., in expressions like $C \bigsqcup D$, where D is not to be understood as the "D in $D^\mathcal{I}$".

With C as a "concept", we have $C^\mathcal{I} \subseteq D^\mathcal{I} \in \mathsf{P}D^\mathcal{I}$. This means that $\mathsf{P}D^\mathcal{I}$ is the actual 'algebra'. *Roles* R are semantically described as relations $R^\mathcal{I} \subseteq D^\mathcal{I} \times D^\mathcal{I}$, i.e., we can equivalently write it as a substitution $R^\mathcal{I} : D^\mathcal{I} \longrightarrow \mathsf{P}D^\mathcal{I}$. The inverse relation R^{-1} is what is actually used on the semantic side, and, in fact, we have

$$(\exists R : C)^\mathcal{I} = \{a \in D^\mathcal{I} \mid \exists (a, b) \in R^\mathcal{I} : b \in C^\mathcal{I}\} = \mu_{D^\mathcal{I}}(\mathsf{P}R^{-1}C).$$

Note how '$\exists$' in $\exists (a, b) \in R^\mathcal{I} : b \in C^\mathcal{I}$ is different from '$\exists$' in $(\exists R : C)^\mathcal{I}$, where in the latter it appears as an informal symbol providing an abstraction of C. In fact, the "existential quantifier" in $\exists R : C$ is an "R-modality" applied to the

powerconcept C. The definition for the semantic expression $(\exists R : C)^{\mathcal{I}}$ uses the existential quantifier that appears in the assumed underlying set theory.

Concerning the underlying signature and related variables, in [6] the situation is unclear, given the assumption about *the existence of two further disjoint alphabets of symbols*, which are called *individual* and *concept variables*. Logically, variables are not part of any alphabet. Variables are terms, and as such they are terms of a certain type. We should therefore speak of "individual concept" rather than "individual variable", and then use x, y, z as variables for individual concepts, and X, Y, Z as variables for concepts.

Now typing of "concept" and "individual concept" comes into play, and we will need type constructors on level two. As opposed to [6], we say "concept" instead of "individual concept", and "powerconcept" instead of "concept". The underlying signature must be formalized, where $\mathtt{concept}$ is a sort in the given underlying signature on level one. On level two, $\mathtt{concept}$ becomes a constant operator, and a type constructor P is then used to produce a new type $\mathsf{P}\mathtt{concept}$, which in their 'algebra' will be understood, respectively, as $D^{\mathcal{I}}$ and $\mathsf{P}D^{\mathcal{I}}$.

Simply typed description logic can now be formally defined in λ-calculus. Let $\Sigma = (S, \Omega)$ be on level one, where $S = \{\mathtt{concept}\}$. Operators in Ω are the constants $\mathtt{c}_1, \ldots, \mathtt{c}_n :\rightarrow \mathtt{concept}$. Concepts and powerconcepts must eventually reside in the same signature on level three. Therefore, on level two, we use $\mathtt{S}_\Sigma$, so that $\mathtt{concept} :\rightarrow \mathtt{type}$ becomes a constant in $\mathtt{S}_\Sigma$. We then include the type constructor $\mathsf{P} : \mathtt{type} \rightarrow \mathtt{type}$ into $\mathtt{S}_\Sigma$, and as the constructed type for "powerconcept". Note that $\mathsf{P}(\mathtt{concept})$ is a term on level two, becoming a sort on level three. A fundamental weakness of traditional description logic is the intertwining of syntax and semantics of the powerconcept. A variable $x \in X_{\mathsf{P}(\mathtt{concept})}$ is a "concept variable" in the sense of [6], and is also a 'term' as an element of $\mathsf{T}_{\Sigma',\mathsf{P}(\mathtt{concept})}(X_{\mathtt{s}})_{\mathtt{s}\in S'}$. On level three we have $\mathtt{c}_1, \ldots, \mathtt{c}_n \in \mathsf{T}_{\Sigma',\mathtt{concept}}(X_{\mathtt{s}})_{\mathtt{s}\in S'}$. "Roles" are of the form $\mathtt{r} :\rightarrow (\mathsf{P}(\mathtt{concept}) \Rrightarrow \mathsf{P}(\mathsf{P}(\mathtt{concept})))$, which creates the need to include operators $\eta :\rightarrow (\mathtt{concept} \Rrightarrow \mathsf{P}(\mathtt{concept}))$ and $\mu :\rightarrow (\mathsf{P}(\mathsf{P}(\mathtt{concept})) \Rrightarrow \mathsf{P}(\mathtt{concept}))$ on level three.

A concept

$$\mathtt{c} :\rightarrow \mathtt{concept}$$

on level one becomes a "singleton powerconcept"

$$\mathtt{app}_{\mathtt{concept},\mathsf{P}(\mathtt{concept})}(\eta, \mathtt{c})$$

on level three, and the syntactic expression "$\exists r.x$" as a term of type $\mathsf{P}(\mathtt{concept})$ can be defined as

$$\exists r.x = \mathtt{app}_{\mathsf{P}(\mathsf{P}(\mathtt{concept})),\mathsf{P}(\mathtt{concept})}(\mu, \mathtt{app}_{\mathtt{concept},\mathsf{P}(\mathsf{P}(\mathtt{concept}))}(\mathtt{r}, x)).$$

"Disjunction" and "negation" are added as new type constructors on level two as

$$\sqcup : \mathsf{P}(\mathtt{concept}) \times \mathsf{P}(\mathtt{concept}) \rightarrow \mathsf{P}(\mathtt{concept})$$

and

$$\neg : \mathsf{P}(\mathtt{concept}) \rightarrow \mathsf{P}(\mathtt{concept}).$$

6 Conclusions

We have shown how syntactic aspects of description logic can be extended to involve generalized relations as compared to just being represented by powerset functors. Double powerset functors, and a range of composed functors can be used, thus representing relations in a more generalized sense. These generalizations are interesting to investigate further over various underlying categories. Viewing many-valued description logic as part of λ-calculus is more general as compared to approaches in [8,7], where fuzzy description logic is basically simply typed description logic with the semantics of P in practice being extended only to the many-valued powerset monad. Our approach reveals the modal nature of description logic more clearly, and shows why it is doubtful to speak about the "existential quantifier" in description logic.

References

1. N. G. De Bruijn, *Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*, Indagationes Mathematicae (1972) 381-392.
2. A. Church, *A formulation of the simple theory of types*, The journal of symbolic logic **5** (1940), 56-68.
3. P. Eklund, M.A. Galán, R. Helgesson, J. Kortelainen, *Fuzzy terms*, Fuzzy Sets and Systems **256** (2014), 211-235.
4. P. Eklund, M.A. Galán, *The rough powerset monad*, Journal of Multiple-Valued Logic and Soft Computing **13** (2007), 321-334.
5. P. Eklund, M.A. Galán, J. Kortelainen, M. Ojeda-Aciego, *Monadic formal concept analysis*, RSCTC 2014, (Eds. C. Cornelis et al.), Lecture Notes in Artificial Intelligence **8536** (2014), 201-210.
6. M. Schmidt-Schauß, G. Smolka, *Attributive concept descriptions with complements*, Artificial Intelligence **48** (1991), 1-26.
7. U. Straccia, *A fuzzy description logic*, in: J. Mostow, C. Rich (Eds.), AAAI/IAAI, AAAI Press / The MIT Press, 1998, 594-599.
8. J. Yen, *Generalizing term subsumption languages to fuzzy logic*, in: IJCAI, 1991, 472-477.