Concepts and Types – An Application to Formal Language Theory

Christian Wurm

Fakultät für Linguistik und Literaturwissenschaften
Universität Bielefeld
cwurm@uni-bielefeld.de

Abstract. We investigate how formal concept analysis can be applied in a type-theoretic context, namely the context of λ-terms typed à la Curry. We first show some general results, which reveal that concept lattices generally respect and reflect the type structure of terms. Then, we show an application of the results in formal language theory, where we vastly generalize existing approaches of capturing the distributional structure of languages by means of formal concept analysis. So type theory is interesting to formal concept analysis not only as a particular context, but also because it allows to generalize existing contexts.

1 Introduction

Formal concept analysis (FCA) operates within what is called a context, that is typically a set of objects, a set of attributes and a relation between them. We will have a closer look at formal concept analysis over terms in a type theoretic context, and show how this can be applied to formal language theory. Our first contribution[1] is that we consider the case not of an atomic typing, but rather recursive, Curry-style typing. Our objects are λ-terms, which have to be assigned types according to their syntactic structure. We show some interesting correlations between the type theoretic structure of sets of terms, their principal typing, and how this these interact with FCA and its lattice-theoretic operations. Our second main contribution is that we show an application to formal language theory. There exist interesting approaches to the distributional structure of languages using FCA over the relation of strings and the contexts in which they occur. However, all of these have some major limitations, as they only work with simple string concatenation, but cannot cope with, for example, the concept of string duplication in a language. We use a type theoretic encoding of strings as terms, and show that this allows us to vastly generalize existing approaches.

2 A Simple Type Theory

Type theory starts with a (usually) finite set of basic types, and a finite, (usually) small set of type constructors. Types are usually interpreted as sets; we denote the

[1] There is considerable work on FCA in a typed context, see, for example, [8]).

Local Proceedings in ISBN 978–84–695–5252–0,
Universidad de Málaga (Dept. Matemática Aplicada), Spain.

set of all objects of type τ by $\|\tau\|$. We will consider only a single type constructor, the usual $\rightarrow$, where for types σ, τ, $\sigma \rightarrow \tau$ is the type of all functions from $\|\sigma\|$ to $\|\tau\|$. Basic objects are assigned some type, and all new objects we can construct in our universe must be constructed in accordance with a typing procedure, that is, we have to make sure that they can be assigned at least one type. Objects which are not well-typed do not exist in the typed universe.

Given a non-empty set A of atomic types, the set of types $Tp(A)$ is defined as closure of A under type constructors: $A \subseteq Tp(A)$, and if $\sigma, \tau \in Tp(A)$, then $\sigma \rightarrow \tau \in Tp(A)$. The **order** of a type is defined as $ord(\sigma) = 0$ for $\sigma \in A$, $ord(\sigma \rightarrow \tau) = max(ord(\sigma) + 1, ord(\tau))$.

We define a higher order signature as $\Sigma := (A, C, \phi)$, where A is a finite set of atomic types, C is a set of constants, and $\phi : C \rightarrow Tp(A)$ assigns types to constants. The order of Σ is $max(\{ord(\phi(c)) : c \in C\})$. Let X be a countable set of variables. The set $\mathtt{Tm}(\Lambda(\Sigma))$, the set of all λ terms over Σ, is the closure of $C \cup X$ under the following rules: 1. $C \cup X \subseteq \mathtt{Tm}(\Lambda(\Sigma))$; 2. if $\mathtt{m}, \mathtt{n} \in \mathtt{Tm}(\Lambda(\Sigma))$, then $(\mathtt{mn}) \in \mathtt{Tm}(\Lambda(\Sigma))$; 3. if $x \in X, \mathtt{m} \in \mathtt{Tm}(\Lambda(\Sigma))$, then $(\lambda x.\mathtt{m}) \in \mathtt{Tm}(\Lambda(\Sigma))$.

We omit the outermost parentheses $(,)$ for λ terms, and write $\lambda x_1 ... x_n.\mathtt{m}$ for $\lambda x_1.(\ldots(\lambda x_n.\mathtt{m})...)$; furthermore, we write $\mathtt{m}_1\mathtt{m}_2\ldots\mathtt{m}_i$ for $(\ldots(\mathtt{m}_1\mathtt{m}_2)\ldots\mathtt{m}_i)$. The set of free variables of a term $\mathtt{m}$, $FV(\mathtt{m})$, is defined by 1. $FV(x) = \{x\} : x \in X$, 2. $FV(c) = \emptyset : c \in C$, 3. $FV(\mathtt{mn}) = FV(\mathtt{m}) \cup FV(\mathtt{n})$, and 4. $FV(\lambda x.\mathtt{m}) = FV(\mathtt{m}) - \{x\}$. $\mathtt{m}$ is closed if $FV(\mathtt{m}) = \emptyset$. We write $\mathtt{m}[\mathtt{n}/x]$ for the result of substituting $\mathtt{n}$ for all free occurrences of x in $\mathtt{m}$. α conversion is defined as $\lambda x.\mathtt{m} \rightsquigarrow_\alpha \lambda y.\mathtt{m}[y/x]$. A β-redex is a term of the form $(\lambda x.\mathtt{m})\mathtt{n}$. We write $\rightsquigarrow_\beta$ for β reduction, so we have $(\lambda x.\mathtt{m})\mathtt{n} \rightsquigarrow_\beta \mathtt{m}[\mathtt{n}/x]$. The inverse of β reduction is β expansion. Let $[\mathtt{m}]_\beta$ denote the β normal form of $\mathtt{m}$, that is, the term without any β redex. This term is unique up to α conversion for every term $\mathtt{m}$. We denote by $=_{\alpha\beta}$ the smallest **congruence** which contains both $\rightsquigarrow_\alpha$ and $\rightsquigarrow_\beta$. We thus write $\mathtt{m} =_{\alpha\beta} \mathtt{n}$, if $\mathtt{n}$ can be derived from $\mathtt{m}$ with any finite series of steps of β-reduction, expansion or α-conversion of any of its subterms.

We now come to the procedure of assigning types to terms.[2] A *type environment* is a (possibly empty) set $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$ of pairs of variables and types, where each variable occurs at most once. A λ-term $\mathtt{m}$ with $FV(\mathtt{m}) = \{x_1, \ldots, x_n\}$ can be assigned a type α in the signature $\Sigma = (A, C, \phi)$ and type environment $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$, in symbols

(1) $\quad x_1 : \alpha_1, \ldots x_n : \alpha_n \vdash_\Sigma \mathtt{m} : \alpha,$

if it can be derived according to the following rules:

(cons) $\vdash_\Sigma c : \phi(c)$, for $c \in C$;

(var) $x : \alpha \vdash_\Sigma x : \alpha$, where $x \in X$ and $\alpha \in Tp(A)$;

[2] We adopt what is known as Curry-style typing: in Church-style typing, terms cannot be constructed without types; in Curry-style typing, terms are first constructed and then assigned a type; so there might be the case that there is no possible assignment.

(abs) $\dfrac{\Gamma \vdash_\Sigma \mathtt{m} : \beta}{\Gamma - \{x : \alpha\} \vdash_\Sigma \lambda x.\mathtt{m} : \alpha \to \beta}$, provided $\Gamma \cup \{x : \alpha\}$ is a type environment;

(app) $\dfrac{\Delta \vdash_\Sigma \mathtt{n} : \alpha \quad \Gamma \vdash_\Sigma \mathtt{m} : \alpha \to \beta}{\Gamma \cup \Delta \vdash_\Sigma \mathtt{mn} : \beta}$, provided $\Gamma \cup \Delta$ is a type environment.

An expression of the form $\Gamma \vdash_\Sigma \mathtt{m} : \alpha$ is called a *judgment*, and if it is derivable by the above rules, it is called the *typing* of $\mathtt{m}$. A term $\mathtt{m}$ is called *typable* if it has a typing. If in a judgment we do not refer to any particular signature, we also write $\Gamma \vdash \mathtt{m} : \alpha$. Regarding β reduction, we have the following well-known result:

Theorem 1 *(Subject Reduction Theorem) If $\Gamma \vdash \mathtt{m} : \alpha$, $\mathtt{m} \rightsquigarrow_\beta \mathtt{m}'$, then $\Gamma' \vdash \mathtt{m}' : \alpha$, where Γ' is the restriction of Γ to $FV(\mathtt{m}')$.*

Let $\mathtt{m} \rightsquigarrow_\beta \mathtt{m}'$ be a contraction of a redex $(\lambda x.\mathtt{n})\mathtt{o}$. This reduction is *non-erasing* if $x \in FV(\mathtt{n})$, and *non-duplicating* if x occurs free in $\mathtt{n}$ at most once. A reduction from $\mathtt{m}$ to $\mathtt{m}'$ is non-erasing (non-duplicating) if all of its reduction steps are non-erasing (non-duplicating). We say a term $\mathtt{m}$ is linear, if for each subterm $\lambda x.\mathtt{n}$ of $\mathtt{m}$, x occurs free in $\mathtt{n}$ exactly once, and each free variable of $\mathtt{m}$ has just one occurrence free in $\mathtt{m}$. Linear λ-terms are thus the terms, for which each β-reduction is non-erasing and non-duplicating. We will be mainly interested in a slightly larger class. A term $\mathtt{m}$ is a λI term, if for each subterm $\lambda x.\mathtt{n}$ of $\mathtt{m}$, x occurs free in $\mathtt{m}$ at least once. λI terms are thus the terms which do not allow for vacuous abstraction (see [1], chapter 9 for extensive treatment). Another important result for us is the following: obviously, by our typing procedure a single term might be possibly assigned many types. We call a type substitution a map $\pi : A \to Tp(A)$, which respects the structure of types: $\pi(\beta \to \gamma) = (\pi(\beta)) \to (\pi(\gamma))$, for $\beta, \gamma \in Tp(A)$.

Theorem 2 *(Principal Type Theorem) Let $\mathtt{m}$ be a term, and let $\Theta := \{\alpha : \Gamma \vdash \mathtt{m} : \alpha$ is derivable$\}$ be the set of all types which can be assigned to $\mathtt{m}$. If $\Theta \neq \emptyset$, then there exists a **principal type** β for $\mathtt{m}$, such that $\Gamma \vdash \mathtt{m} : \beta$ is derivable, and for each $\alpha \in \Theta$, there is a substitution π_α such that $\alpha = \pi_\alpha(\beta)$.*

Obviously, β is unique up to isomorphism; we will write $pt(\mathtt{m})$ for the principal type of $\mathtt{m}$. The proof of the theorem is constructive, that is, β can be effectively computed or shown to be nonexistent, see [6].

3 Types and Concepts

3.1 A Context of Terms

We now give a short introduction into formal concept analysis. A context is a triple $(\mathcal{G}, \mathcal{M}, I)$, where $\mathcal{G}, \mathcal{M}$ are sets and $I \subseteq \mathcal{G} \times \mathcal{M}$. In FCA, the entities in $\mathcal{G}$ are thought of as objects, the objects in $\mathcal{M}$ as attributes, and for $m \in \mathcal{M}$, $g \in \mathcal{G}$, we have $(g, m) \in I$ if the object g has the attribute m. This is all we need as basic structure to get the machine of FCA going. For $A \subseteq \mathcal{G}, B \subseteq \mathcal{M}$, we put $A^\triangleright := \{m \in \mathcal{M} : \forall a \in A,\ (a, m) \in I\}$, and $B^\triangleleft := \{g \in \mathcal{G} : \forall m \in B,\ (a, m) \in I\}$.

A concept is a pair (A, B) such that $A^{\triangleright} = B, B^{\triangleleft} = A$. We call A the **extent** and B the **intent**. A is the extent of a concept iff $A = A^{\triangleright\triangleleft}$, dually for intents. The maps $[-]^{\triangleright}, [-]^{\triangleleft}$ are called **polar maps**. We order concepts by inclusion of extents, that is, $(A_1, B_1) \leq (A_2, B_2) \Leftrightarrow A_1 \subseteq A_2$.

Definition 3 *Given a context $\mathfrak{B} = (\mathcal{G}, \mathcal{M}, I)$, we define the concept lattice of $\mathfrak{B}$ as $\mathcal{L}(\mathfrak{B}) = \langle \mathfrak{C}, \wedge, \vee, \top, \bot \rangle$, where $\top = (\mathcal{G}, \mathcal{G}^{\triangleright})$, $\bot = (\mathcal{M}^{\triangleleft}, \mathcal{M})$, and for $(A_i, B_i), (A_j, B_j) \in \mathfrak{C}$, $(A_i, B_i) \wedge (A_j, B_j) = (A_i \cap A_j, (B_i \cup B_j)^{\triangleleft\triangleright})$, and $(A_i, B_i) \vee (A_j, B_j) = ((A_i \cup A_j)^{\triangleright\triangleleft}, B_i \cap B_j)$.*

We define our type theoretic context as follows. Recall that $\mathtt{Tm}(\Lambda(\Sigma))$ is the set of all λ-terms over Σ. We put $\mathtt{Tm}_c(\Lambda(\Sigma)) := \{\mathtt{m} \in \mathtt{Tm}(\Lambda(\Sigma)) : FV(\mathtt{m}) = \emptyset\}$, the set of closed terms, and we call $\mathtt{Tm}_c(\Lambda I(\Sigma))$ the set of all closed λI terms. Furthermore, define $\mathtt{WTT}$ as the set of all closed and well-typed terms, that is, the set of all terms $\mathtt{m}$ such that $\vdash \mathtt{m} : \alpha$ is derivable for some α by our rules; $\mathtt{WTT}_I = \mathtt{WTT} \cap \mathtt{Tm}(\Lambda I(\Sigma))$. Recall that $=_{\alpha\beta}$ is a congruence. Let σ be a given type, and $L' \subseteq \|\sigma\|$ be a distinguished subset of the terms of type σ; we define $L := \{\mathtt{m} : \exists \mathtt{n} \in L' : \mathtt{m} =_{\alpha\beta} \mathtt{n}\}$, that is, as closure of L' under $=_{\alpha\beta}$. Put $\mathcal{G} = \mathcal{M} = \mathtt{Tm}_c(\Lambda(\Sigma))$, and define the relation $I \subseteq \mathtt{Tm}_c(\Lambda(\Sigma)) \times \mathtt{Tm}_c(\Lambda(\Sigma))$ as follows: for $\mathtt{m}, \mathtt{n} \in \mathtt{Tm}_c(\Lambda(\Sigma))$, we have $(\mathtt{m}, \mathtt{n}) \in I$ if $\mathtt{nm} \in L$. So the relation of objects in $\mathcal{M}$ and $\mathcal{G}$ is that of function and argument, and the relation I tells us whether the two yield a desired value. Same can be done with $\mathtt{Tm}_c(\Lambda I(\Sigma))$.

Obviously, we have $\bot = (\emptyset, \mathtt{Tm}_c(\Lambda(\Sigma)))$. Regarding upper bounds, we have to distinguish two important concepts: we first have a concept we denote $\mathtt{T} := (\mathtt{WTT}, \Lambda V)$, where ΛV (V for vacuous) is the set of all terms of the form $\lambda x.\mathtt{m}$, where $\mathtt{m} \in L$ and $x \notin FV(\mathtt{m})$. There is however a larger concept $\top \geq \mathtt{T}$, which is defined as $\top := (\mathtt{Tm}_c(\Lambda(\Sigma)), \emptyset)$. The reason for this slight complication is as follows: we want our terms to be closed, because open terms are meaningless for us. Now, it holds that the concatenation of closed terms is again a closed term; but the concatenation of well-typed terms need not be well-typed: for $\mathtt{m}, \mathtt{n} \in \mathtt{WTT}$, it might be that $\mathtt{nm} \notin \mathtt{WTT}$. Furthermore, there are λ-terms $\mathtt{n}$ with vacuous abstraction such that the set $\{\mathtt{nm} : \mathtt{m} \in \mathtt{T}\} \lneq \mathtt{T}$; we would however like our $\top$ to be absorbing; and in fact, if $\mathtt{m} \notin \mathtt{WTT}$, then for any term $\mathtt{n}$, $\mathtt{nm}, \mathtt{mn} \notin \mathtt{WTT}$. So for every term $\mathtt{m} \notin \mathtt{WTT}$, $\{\mathtt{m}\}^{\triangleright} = \emptyset$. For all interesting results we have to restrict ourselves to $\mathtt{WTT}$, but for completeness of some operations we have to consider $\mathtt{Tm}_c(\Lambda(\Sigma))$. Note however that if we restrict $\mathtt{Tm}_c(\Lambda(\Sigma))$ to $\mathtt{Tm}_c(\Lambda I(\Sigma))$, then $\top$ and $\mathtt{T}$ coincide.[3]

3.2 Concept Structure and Type Structure

We have seen that each term can be assigned a most general type. Importantly, the same holds for sets of types:

[3] This is actually not straightforward, but follows as a corollary from results we present later on.

Lemma 4 *Let $T \subseteq \mathtt{WTT}$ be a set of terms, such that the set of principal types $\{pt(\mathtt{m}) : \mathtt{m} \in T\}$ is finite. If there is a set of types Θ, such that for each $\mathtt{m} \in T$ and all $\theta \in \Theta$, $\vdash \mathtt{m} : \theta$ is a derivable judgment, then there is a (up to isomorphism) unique type α, such that for every $\mathtt{m} \in T$, $\vdash \mathtt{m} : \alpha$ is derivable, and every $\theta \in \Theta$ can be obtained by α through a type substitution.*

α is usually called the **most general unifier** of Θ; for a set of terms T, we also directly call it $pt(T)$, the principal type of T; for Θ a set of types, we denote it by $\bigvee \Theta$. A proof for this fundamental lemma can be found in [6]; again the proof is constructive. Note that if we do not assume that the set of principal types of terms $\mathtt{m} \in T$ is finite, then there is no upper bound on the length of types, and so there cannot be a finite common unifier. For convenience, we introduce an additional type $\top \notin A$, such that our types are the set $\{\top\} \cup Tp(A)$. If a set of types Θ does not have a common unifier, then we put $\bigvee(\Theta) = \top$.

This is of immediate importance for us, as it allows us both to speak of the principal type of a set of terms, as well as of the least upper bound of a set of types. From there we easily arrive at the greatest lower bound of two types α, β, which we denote by $\alpha \wedge \beta$, and which intuitively is the amount of structure which α and β share. Write $\alpha \leq \beta$, if there is a substitution π such that $\pi(\alpha) = \beta$. This is, up to isomorphism, a partial order. We now can simply define $\alpha \wedge \beta := \bigvee\{\gamma : \gamma \leq \alpha, \beta\}$. It is clear that the set $\{\gamma : \gamma \leq \alpha, \beta\}$ modulo isomorphism is finite, so the (finite) join exists in virtue of the above lemma. So $Tp(A)$ is lattice ordered up to isomorphism.

How does type structure behave wrt. concept structure? First of all, if $A \subseteq B$, then $pt(A) \leq pt(B)$. So the inclusion relation reflects type structure. This entails that $pt(A) \leq pt(B^{\triangleright\triangleleft})$. Stronger results are hard to obtain; for example, if we know $pt(A)$, there is nothing we can say in general about an upper bound for $pt(A^{\triangleright\triangleleft})$.

Fortunately, there is more we can say about the lattice order of concepts and type order. Define $\vee$ and $\wedge$ on concepts as usual. For a concept (A, B) over the term context, we put $pt_1(A, B) = pt(A)$, $pt_2(A, B) = pt(B)$.

Lemma 5 *For concepts $\mathcal{C}_1, \mathcal{C}_2$ of the term context, the following holds: (1) If $\mathcal{C}_1 \leq \mathcal{C}_2$, then $pt_1(\mathcal{C}_1) \leq pt_1(\mathcal{C}_2)$, and $pt_2(\mathcal{C}_2) \leq pt_2(\mathcal{C}_1)$. (2) $pt_1(\mathcal{C}_1 \wedge \mathcal{C}_2) \leq pt_1(\mathcal{C}_1) \wedge pt_1(\mathcal{C}_2)$, and (3) $pt_1(\mathcal{C}_1) \vee pt_1(\mathcal{C}_2) \leq pt_1(\mathcal{C}_1 \vee \mathcal{C}_2)$.*

Proof. The first claim is immediate by set inclusion. To see the second, consider that for every $\mathtt{m} \in A_1 \cap A_2$, we must have $pt(\{\mathtt{m}\}) \leq pt(A_1), pt(\{\mathtt{m}\}) \leq pt(A_2)$ by set inclusion; and so $pt(\{\mathtt{m}\}) \leq pt_1(C_1) \wedge pt_1(\mathcal{C}_2)$. To see the third claim, consider the following: we can easily show that $pt(A_1) \vee pt(A_2) = pt(A_1 \cup A_2)$. Then the claim follows from considering that $pt(A_1 \cup A_2) \leq pt((A_1 \cup A_2)^{\triangleright\triangleleft})$. □

Definition 6 *A term $\mathtt{m}$ is a **left equalizer**, if we have $\vdash \mathtt{m} : \theta_1 \to \alpha$, $\vdash \mathtt{m} : \theta_2 \to \alpha$, and $\theta_1 \neq \theta_2$. $\mathtt{m}$ is a **right equalizer**, if $\vdash \mathtt{m} : \alpha_1$, $\vdash \mathtt{m} : \alpha_2$, and $\alpha_1 \neq \alpha_2$.*

Easy examples of left equalizers are terms with vacuous abstraction; easy examples of right equalizers are terms which do not contain constants. A term

which is both a left and right equalizer is $\lambda yx.x$. The following results are a bit tedious to obtain, yet not very significant; we therefore omit the proof.

Lemma 7 *Let $T \subseteq \mathtt{WTT}$, such that $pt(T) = \top$. Then each $\mathtt{m} \in T^{\triangleright}$ is a left equalizer.*

We can thus also speak of equalizer concepts. If we restrict our context to λI terms, we get a stronger result:

Lemma 8 *Let $\mathtt{m}$ be a left equalizer and λI-term, such that $\vdash_\Sigma \mathtt{m} : \theta_1 \to \alpha$ and $\vdash_\Sigma \mathtt{m} : \theta_2 \to \alpha$. Then both θ_1, θ_2 must be types inhabited by terms in $\mathtt{Tm}(\Lambda I(\Sigma))$, that is, there are terms $\mathtt{m}_i$, for which $\vdash_\Sigma \mathtt{m}_i : \theta_i$ is derivable for $i \in \{1, 2\}$ and $\mathtt{m}_i \in \mathtt{Tm}(\Lambda I(\Sigma))$.*

So when we restrict ourselves to ΛI, we have proper restrictions on the class of possible equalizers, in the general case we do not. For example, assume there is a set T of terms, and $pt(T) \neq \top$. Still, we might have $pt(T^{\triangleright}) = \top$. Conversely, from the fact that $pt(T) = \top$, it does not follow that $T^{\triangleright} = \emptyset$.

Of course, all general results of FCA also hold in this particular setting. For us, the question is not in how far is the type theoretic context interesting as a *particular* context, but rather: in how far can type theoretic contexts be used in order to *generalize* existing contexts? As is well-known, type theory is a very powerful tool; we will show this by way of example in formal language theory.

4 A Language-theoretic Context

4.1 Syntactic Concepts

Syntactic concept lattices form a particular case of formal concept lattices. In linguistics, they have been introduced in [9]. They were brought back to attention and enriched with residuation in [2], [3], as they turn out to be useful representations for language learning (for background on residuated lattices, see [5]). Syntactic concept are useful to describe distributional patterns of strings[4]. The most obvious way to do so is by partitioning strings/substrings into *equivalence classes*: we say that two strings w, v are equivalent in a language $L \subseteq T^*$, in symbols, $w \sim_L v$, iff for all $x, y \in T^*$, $xwy \in L \Leftrightarrow xvy \in L$.[5] The problem with equivalence classes is that they are too restrictive: a single word can ruin an equivalence class. In particular in linguistic applications, this is bad, because restrictions of datasets or some particular constructions might prevent us from having, say, an equivalence class of nouns. Syntactic concepts provide a somewhat less rigid notion of equivalence, which can be conceived of as equivalence restricted to a given set of string-contexts (not to be confused with contexts in the sense of FCA!).

[4] Or words, respectively, depending on whether we think of our language as a set of words or a set of strings of words.

[5] This defines the well-known Nerode-equivalence.

We now define our first language theoretic context. For $L \subseteq T^*$, $\mathfrak{B}_C(L) = (T^*, T^* \times T^*, I)$, where $(b,(a,c)) \in I$ iff $abc \in L$. This gives rise to polar maps $\triangleright : \wp(T^*) \to \wp(T^* \times T^*)$, and $\triangleleft : \wp(T^* \times T^*) \to \wp(T^*)$, where

1. for $S \subseteq T^*$, $S^{\triangleright} := \{(x,y) : \forall w \in S, xwy \in L\}$; and dually
2. for $C \subseteq T^* \times T^*$, $C^{\triangleleft} := \{x : \forall (v,w) \in C, vxw \in L\}$.

That is, a set of strings is mapped to the set of string contexts, in which all of its elements can occur. For a set of string contexts C, $C^{\triangleleft}$ can be thought of as an equivalence class with respect to the string contexts in C; but not in general: there might be elements in $C^{\triangleleft}$ which can occur in a string context $(v,w) \notin C$ (and conversely).

Definition 9 *A syntactic c-concept A is a pair, consisting of a set of strings, and a set of string contexts, written $\mathcal{C} = (S_{\mathcal{C}}, C_{\mathcal{C}})$, such that $S_{\mathcal{C}}^{\triangleright} = C_{\mathcal{C}}$ and $C_{\mathcal{C}}^{\triangleleft} = S_{\mathcal{C}}$. The* ***syntactic c-concept lattice*** *of a language L is defined as $\mathcal{L}(\mathfrak{B}_C(L)) := \langle \mathfrak{C}_L^C, \wedge, \vee, \top, \bot \rangle$, where $\mathfrak{C}_L^C$ is the set of syntactic c-concepts of L, and with all constants and connectors defined in the usual way.*

For example, given a language L, we have $(\epsilon,\epsilon)^{\triangleleft} = L$, as all and only the strings in L can occur in L in the string context (ϵ,ϵ); so L is a closed set of strings. We can give the syntactic concept lattice some more structure. We define a monoid structure on concepts as follows: for concepts $(S_1,C_1),(S_2,C_2)$, we define:

$$(2) \qquad (S_1,C_1) \circ (S_2,C_2) = ((S_1S_2)^{\triangleright\triangleleft}, (S_1S_2)^{\triangleright}),$$

where $S_1S_2 = \{xy : x \in S_1, y \in S_2\}$. Obviously, the result is a concept. $'\circ'$ is associative on concepts, that is, for $X,Y,Z \in \mathfrak{B}$, $X \circ (Y \circ Z) = (X \circ Y) \circ Z$ (see [10] for discussion). It is easy to see that the neutral element of the monoid is $(\{\epsilon\}^{\triangleright\triangleleft}, \{\epsilon\}^{\triangleright})$, and that the monoid structure respects the partial order of the lattice: For concepts $X,Y,Z,W \in \mathfrak{B}$, if $X \leq Y$, then $W \circ X \circ Z \leq W \circ Y \circ Z$.

We define a similar operation $\bullet$ for the string contexts of concepts: $(x,y) \bullet (w,z) = (xw, zy)$. This way, we still have $f \bullet (g \bullet h) = (f \bullet g) \bullet h$ for singleton string contexts f,g,h. The operation can be extended to sets in the natural way, preserving associativity. For example, $C \bullet (\epsilon, S) = \{(x,ay) : (x,y) \in C, a \in S\}$. We will use this as follows:

Definition 10 *Let $X = (S_X, C_X), Y = (S_Y, C_Y)$ be concepts. We define the right residual $X/Y := ((C_1 \bullet (\epsilon, S_Y))^{\triangleleft}, (C_1 \bullet (\epsilon, S_Y))^{\triangleleft\triangleright})$, and the left residual $Y\backslash X := ((C_1 \bullet (S_Y, \epsilon))^{\triangleleft}, (C_1 \bullet (S_Y, \epsilon))^{\triangleleft\triangleright})$.*

For the closed sets of strings S,T, define $S/T := \{w : \text{for all } v \in T, wv \in S\}$. We then have $S_X/S_Y = S_{X/Y}$. So residuals are unique and satisfy the following lemma:

Lemma 11 *For $X,Y,Z \in \mathfrak{C}_L^C$, we have $Y \leq X\backslash Z$ iff $X \circ Y \leq Z$ iff $X \leq Z/Y$.*

For a proof, see [2]. This shows that the syntactic concept lattice can be enriched to a residuated lattice (a residuated lattice is precisely a lattice with monoid structure and satisfying the law of residuation).

4.2 Problems and Limitations

Syntactic c-concepts form a very well-behaved structure, which even forms a complete class of models for the Full Lambek calculus, an important substructural logic (see [10]). A major limitation of these concepts is that our only objects are strings, and our only operation concatenation. To see why this is a restriction, consider the following. Let $L_1 \subseteq T^*$ be an arbitrary language, and put $L_2 := \{ww : w \in L_1\}$. Now, in the general case, L_1 will not be a closed concept of L_2, because there is no general string context in which all and only the words in L_1 can occur. The appropriate string context would have to be a *function*, as each $w \in L_1$ has the string context (ϵ, w). So there is a clear and simple generalization regarding the distribution of L_1 in L_2, but we cannot express it.

As a second example, consider $L_3 := \{a^{2^n} : n \in \mathbb{N}\}$. Here we have the following problem: for each $n \in \mathbb{N}$, a^n will have a distinguishing string context. Nonetheless, there is a very simple pattern in the language: taking a word of L_3, we just have to concatenate it with itself, and we get a new word in L_3. In our analysis, however, there are no concepts $\mathcal{C}_1, \mathcal{C}_2$, such that $\mathcal{C}_1 \circ (L_3, L_3^{\triangleright}) \circ \mathcal{C}_2 = (L_3, L_3^{\triangleright})$. So our concepts are uninformative on the pattern of this language. What we want to have in this case is a concept of *duplication*. We will remedy these shortcomings in what is to follow; we will need, however, some type-theoretic background.

5 Strings as λ-Terms

The following, type theoretic encoding of language theoretic entities has been developed in the framework on on **abstract categorial grammars** (introduced in [4]). We follow the standard presentation given in [7]. Given a finite alphabet T, a string $a_1 \dots a_n \in T^*$ over T can be represented by a λ term over the signature $\Sigma_T^{string} := (\{o\}, T, \phi)$, where for all $a \in T$, $\phi(a) = o \to o$; we call this a *string signature*. The term is linear and written as $/a_1 \dots a_n/ := \lambda x.a_1(\dots(a_n x)\dots)$. Obviously, the variable x has to be type o, in order to make the term typable. We then have, for every string $w \in T^*$, $\vdash_{\Sigma_T^{string}} /w/ : o \to o$.

Under this representation, string concatenation is not entirely trivial, and cannot be done by juxtaposition, as the result would not be typable. We can concatenate strings by the combinator $\mathbf{B} := \lambda xyz.x(yz)$, which concatenates its first argument to the left of its second argument, as can be easily checked.[6] We can also represent tuples of strings by terms. Let $/w_1/, \dots, /w_n/$ represent strings. Then a tuple of these strings is written as $/(w_1, \dots, w_n)/ := \lambda x.((\dots(x/w_1/)\dots)/w_n/)$. The type of x here depends on the size of the tuple. We define $\alpha \to_n \beta$ by $\alpha \to_0 \beta = \beta$, $\alpha \to_{n+1} \beta = \alpha \to (\alpha \to_n \beta)$. In general, for a term $\mathtt{m}$ encoding an n-tuple , we have $\vdash_{\Sigma_T^{string}} \mathtt{m} : ((o \to o) \to_n (\alpha))) \to \alpha$. So the types get larger with the size of tuples; the *order* of the term however remains invariantly 2.

We indicate how to manipulate tuple components separately. The function which concatenates the tuple components in their order is obtained as

[6] See [7] for more examples, also for what is to follow. A *combinator* is in general a function over functions.

follows: Given a tuple $/(w,v)/ = \lambda x.((x/w/)/v/$, we obtain $/wv/$ through application of the term: $\lambda x_1.x_1(\lambda x_2 y.\mathbf{B}x_2 y))$. We can also manipulate tuples to form new tuples: take again $/(v_1,w_1)/ = \lambda x.((x/v_1/)/w_1/)$; we want to convert it into a tuple $/(v_1v_2,w_1w_2)/ = \lambda x.((x/v_1v_2/)/w_1w_2/)$. This is done by the term $\lambda y x_1.y(\lambda x_2 x_3((x_1\mathbf{B}x_2v_2)\mathbf{B}x_3w_2))$. This term takes the tuple as argument and returns a tuple of the same type. If we abstract over the term $/(v_2,w_2)/$, this gives us a function which concatenates two 2-tuples componentwise.

However, the general componentwise concatenation of tuples of arbitrary size (considering strings as 1-tuples) cannot be effected by a typed λ-term. The reason is: if we do not fix an upper bound on tuple size, the types of tuples get higher and higher, and there is no finite upper bound. So there is no finite term which could have the appropriate type.[7] This means that in this setting, we must refrain from a notion of general concatenation of any type. This will however do little harm, as we will see.

6 Generalizing the Language-theoretic Context

Take a finite alphabet T, and fix a language $L_1 \subseteq T^*$. As we have seen in the last section, there is a bi-unique mapping $i : T^* \to \mathtt{WTT}$ between strings in T^* and λ-terms of the signature Σ_T^{string}. Note that i is properly bi-unique and not up to $=_{\alpha\beta}$ equivalence; we map strings only onto their standard encoding, using a standard variable. We thus obtain $i[L_1] \subseteq \mathtt{WTT}$, where $i[-]$ is the pointwise extension of i to sets. We close $i[L_1]$ under $=_{\alpha\beta}$, and obtain $L := \{\mathtt{m} : \text{there is } \mathtt{n} \in i[L] : \mathtt{n} =_{\alpha\beta} \mathtt{m}\}$. This is the language we are working with, the type theoretic counterpart of L_1. In the sequel, for any $M \subseteq T^*$, we will denote the closure of $i[M]$ under $=_{\alpha\beta}$ by M^λ; so we have $L = (L_1)^\lambda$.

We now define a context $\mathfrak{B}_T(L) = (\mathcal{G}, \mathcal{M}, I)$, where $\mathcal{G} = \mathcal{M} = \mathtt{Tm}_c(\Lambda(\Sigma_T^{string}))$, that is the set of closed terms over the signature Σ_T^{string}; and for $\mathtt{m},\mathtt{n} \in \mathtt{Tm}_c(\Lambda(\Sigma_T^{string}))$, we have $(\mathtt{m},\mathtt{n}) \in I$ iff $\mathtt{nm} \in L$. So for S a set of terms, we have $S^\triangleright := \{t : \forall s \in S : ts \in L\}$, and $S^\triangleleft := \{t : \forall s \in S : st \in L\}$.

Definition 12 *A t-concept is a concept (S,T) over the context $\mathfrak{B}_T(L)$, where $S = T^\triangleleft$, $T = S^\triangleright$. The* ***syntactic t-concept lattice*** *of a language L is defined as $\mathcal{L}_T(L) := \mathcal{L}(\mathfrak{B}_T(L)) = \langle \mathfrak{C}_L^T, \wedge, \vee, \top, \bot\rangle$, where $\mathfrak{C}_L^T$ is the set of syntactic t-concepts of L, and with all constants and connectors defined in the usual way.*

What we are still missing is an operator which allows us to define fusion and residuation. Recall that for terms, our primitive objects, juxtaposition is interpreted as function application. We extend this interpretation to sets of terms: for $S_1, S_2 \subseteq \mathtt{Tm}_c(\Lambda(\Sigma_T^{string}))$, we define $S_1S_2 := \{\mathtt{mn} : \mathtt{m} \in S, \mathtt{n} \in T\}$. Next, for t-concepts $(S_1,T_1),(S_2,T_2)$, we simply put $(S_1,T_1)\circ(S_2,T_2) := ((S_1S_2)^{\triangleright\triangleleft}, (S_1S_2)^\triangleright)$.

[7] On the other side, once we fix an upper bound k to tuple size, it is easy to see how to define $\circ$ as λ term: for $i \leq k$, we simply encode all tuples as k-tuples with all jth components, $i < j$, containing the empty string. Then $\circ$ is simply componentwise concatenation of k-tuples, which is λ-definable, as we have seen.

That is, as before we use the closure of concatenation of extents to define $\circ$. But there is an important restriction: concatenation of terms is *not* associative. Consequently, the operation $\circ$ is not associative on concepts, we have, for concepts $M, N, O \in \mathfrak{C}_L^T$, $(M \circ N) \circ O \neq M \circ (N \circ O)$. For example, $M \circ N$ might be $\top$, because MN contains a term $\mathtt{mn} \notin \mathtt{WTT}$, and consequently we have $\top \circ O = \top$. Still, $M \circ (N \circ O)$ might be well-typed. So the structure of $(\mathfrak{B}, \circ)$ is not a monoid, but rather a groupoid. We furthermore have a left identity element 1_l, such that for every concept S, $1_l \circ S = S$. This is the concept of the identity function $(\{\lambda x.x\}^{\triangleright\triangleleft}, \{\lambda x.x\}^{\triangleright})$. (By the way, the identity function is also the encoding of the empty string $/\epsilon/$). There is no general right identity, though: for assume we have a term $\mathtt{m} : \alpha$ for a constant atomic type α; then there is no term $\mathtt{n}$ such that $\mathtt{mn}$ can be typed. Consequently, no $\mathtt{n}$ can be the right identity for $\mathtt{m}$.

What are the residuals in this structure? Given the fusion operator, they are already implicitly defined by the law of residuation $O \leq M/N \Leftrightarrow O \circ N \leq M \Leftrightarrow N \leq O\backslash M$; what we have to show that they exist and are unique. In the sequel we will use residuals both on sets of terms and on concepts; this can be done without any harm, as the extent order and the concept order are isomorphic. To see more clearly what residuation means in our context, note that for $S \subseteq \mathtt{Tm}_c(\Lambda(\Sigma_T^{string}))$, we have $S^{\triangleright} := L/S$; because $S^{\triangleright}$ is the set of all terms $\mathtt{m}$, such that for all $\mathtt{n} \in S$, $\mathtt{mn} \in L$. Dually, we have $S^{\triangleleft} := S\backslash L$. Consequently, we have $S^{\triangleright\triangleleft} = (L/S)\backslash L$, and dually, we get $S^{\triangleleft\triangleright} = L/(S\backslash L)$. So we see that the polar maps of our Galois connection form a particular case of the residuals, or conversely, the residuals form a generalization of the polar maps. The closure operators are equivalent to a particular case of what is known as *type raising*. More generally, we can explicitly define residuals over a ternary relation: put $(\mathtt{m}, \mathtt{n}, \mathtt{o}) \in R$ if and only if $\mathtt{mn} =_{\alpha\beta} \mathtt{o}$. Then we define

1. $O/N := \{\mathtt{m} : \forall \mathtt{n} \in N, \exists \mathtt{o} \in O : (\mathtt{m}, \mathtt{n}, \mathtt{o}) \in R\}$; dually:
2. $M\backslash O := \{\mathtt{n} : \forall \mathtt{m} \in M, \exists \mathtt{o} \in O : (\mathtt{m}, \mathtt{n}, \mathtt{o}) \in R\}$.

As is easy to see, $M^{\triangleright} := \{\mathtt{n} : \forall \mathtt{m} \in M, \exists \mathtt{o} \in L : (\mathtt{m}, \mathtt{n}, \mathtt{o}) \in R\}$; and $M^{\triangleleft} := \{\mathtt{n} : \forall \mathtt{m} \in M, \exists o \in L : (\mathtt{m}, \mathtt{n}, \mathtt{o}) \in R\}$. This way, we explicitly define residuals for sets of terms. Given this, it easily follows that residuals also exist and are unique for concepts: $(S_1, T_1)/(S_2, T_2) = ((S_1/S_2), (S_1/S_2)^{\triangleright})$.

So residuals allow us to form the closure not only with respect to L, but with respect to any other concept. This provides us with a much more fine-grained access to the hierarchical structure of languages. On the negative side, the $\circ$ operation and residuals do not tell us anything about directionality of concatenation on the string level. This however is unsurprising, as our treatment of strings as λ-terms serves precisely the purpose of abstracting away from this: concatenation is done by terms automatically, and we need no longer care for this. Obviously t-concepts provide a vast generalization of c-concepts. An immediate question is whether this extension is conservative, in the sense that each c-closed set is also t-closed. This is generally wrong, but holds with some restrictions:

Theorem 13 *Let $M, L \subseteq T^*$; let M^{λ}, L^{λ} be their type theoretic counterpart in the signature Σ_T^{string}. If $M = M^{\triangleright\triangleleft}$ is closed wrt. the language theoretic context*

$\mathfrak{B}_C(L)$, then we have $M^\lambda = (M^\lambda)^{\triangleright\triangleleft} \cap (T^*)^\lambda$, where $(M^\lambda)^{\triangleright\triangleleft}$ is closed wrt. the type theoretic context $\mathfrak{B}_T(L^\lambda)$.

Proof. Let M be c-closed; every string context $(w, v) \in M^\triangleright$ corresponds to a function of the form $\lambda x.\mathbf{B}(\mathbf{B}/w/x)/v/$, which takes a term $/u/$ as argument, concatenating it with a $/w/$ to its left and $/v/$ to its right, resulting in a term $/wuv/$. Call the set of these functions $(M^\lambda)^\blacktriangleright$. We now take $(M^\lambda)^{\blacktriangleright\triangleleft}$. Obviously we have $M^\lambda \subseteq (M^\lambda)^{\blacktriangleright\triangleleft}$. We show that $M^\lambda \supseteq (M^\lambda)^{\blacktriangleright\triangleleft} \cap (T^*)^\lambda$: if we have, for $w \in T^*$, $w \notin M$, but $/w/ \in (M^\lambda)^{\blacktriangleright\triangleleft} \cap (T^*)^\lambda$, then we have $i^{-1}(/w/) \in M^{\triangleright\triangleleft}$, because each type context in $(M^\lambda)^\blacktriangleright$ corresponds to a string context in $M^\triangleright$. This is a contradiction, as M is closed under $[-]^{\triangleright\triangleleft}$.

So we have $M^\lambda = (M^\lambda)^{\blacktriangleright\triangleleft} \cap (T^*)^\lambda$, and $(M^\lambda)^{\blacktriangleright\triangleleft}$ is a closed set. Furthermore, as $(M^\lambda)^\blacktriangleright \subseteq (M^\lambda)^\triangleright$, we have (by the laws of Galois connections) $(M^\lambda)^{\blacktriangleright\triangleleft} \supseteq (M^\lambda)^{\triangleright\triangleleft}$. So we get $M^\lambda \supseteq (M^\lambda)^{\triangleright\triangleleft} \cap (T^*)^\lambda$. To see that $M^\lambda \subseteq (M^\lambda)^{\triangleright\triangleleft} \cap (T^*)^\lambda$, consider that as $M \subseteq T^*$, we have $M^\lambda \subseteq (T^*)^\lambda$; furthermore, $M^\lambda \subseteq (M^\lambda)^{\triangleright\triangleleft}$. Therefore, $M^\lambda \subseteq (M^\lambda)^{\triangleright\triangleleft} \cap (T^*)^\lambda$. This completes the proof. □

As expected, the converse does not hold, not even for terms which encode strings. In this sense t-concepts yield a proper generalization of c-concepts. This however does not obtain for the extension of the lattice with fusion and residuals: fusion in the t-concept lattice is completely incomparable to fusion in the c-concept lattice of a language.

7 Conclusion and Possible Restrictions

One main objection to our type theoretic approach to language might be that we produce many concepts to which we might not be able to assign any intuitive meaning, and which tell us very little about the language in question. We easily see what is meant by the concept of a term $/w/$ in a language L. We can also make perfectly sense of the concept duplication. It is less easy to see what is meant by the concept of the term $\mathbf{B}$, which we discussed above. What can the the distribution of such a concept in a language tell us about the language?[8] So we do not have a problem with the formalism in the first place, but with its interpretation.

Therefore, it might be reasonable to restrict our approach. We propose here two main restrictions: First, as we already mentioned, we might restrict the universe of terms $\mathtt{Tm}_c(\Lambda(\Sigma_T^{string}))$ to λI terms. A language-theoretic argument for this point is that we are interested in the distributional structure of languages. Vacuous abstraction, as yielding constant functions, allows us to delete arguments or certain parts thereof. This seems to us an "unlinguistic" procedure, as we cannot say we talk about the distribution of an object if we allow to delete parts of it. A further restriction to linear λ-terms, on the other side, does not

[8] What is less unclear is its meaning as intent rather than extent: apart from some additional technical difficulties, it must take two arguments which, when concatenated, give a term in L.

seem to be desirable, as then we get the same problems with our toy language $\{a^{2^n} : n \in \mathbb{N}\}$ as before.

A second restriction which might be reasonable is the restriction of **type order**. As we have seen, types of second order allow us to yield all strings and tuples of strings. If we restrict only $\mathcal{G}$ to second order types, we will have all functions from second order types to second order types in $\mathcal{M}$. This seems to us a very reasonable restriction, the consequence of which we cannot discuss here for reasons of space.

In conclusion, there are many options in further pursuing our approach, and at this point it is unclear which direction is the most promising. But in either way our approach might provide some contribution to the old problem of learning infinite languages from the distributional structure of a finite fragment thereof.

References

1. Henk Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Number 103 in Studies in Logic. Elsevier, Amsterdam, 2 edition, 1985.
2. Alexander Clark. A learnable representation for syntax using residuated lattices. In Philippe de Groote, Markus Egg, and Laura Kallmeyer, editors, *Proceedings of the 14th Conference on Formal Grammar*, volume 5591 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2009.
3. Alexander Clark. Learning context free grammars with the syntactic concept lattice. In José M. Sempere and Pedro García, editors, *10th International Colloquium on Grammatical Inference*, volume 6339 of *Lecture Notes in Computer Science*, pages 38–51. Springer, 2010.
4. Philippe de Groote. Towards Abstract Categorial Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter*, pages 148–155, Toulouse, 2001.
5. Nikolaos Galatos, Peter Jipsen, Tomasz Kowalski, and Hiroakira Ono. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*. Elsevier, 2007.
6. J.Roger Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 2008.
7. Makoto Kanazawa. Second-order Abstract Categorial Grammars as Hyperedge Replacement Grammars. *Journal of Logic, Language and Information*, 19(2):137–161, 2010.
8. Ian D. Peake, Ian E. Thomas, and Heinz W. Schmidt. Typed formal concept analysis. In Karl Erich Wolff, Sebastian Rudolph, and Sebastien Ferre, editors, *Contributions to ICFCA 2009 (Supplementary Proceedings), 7th International Conference on Formal Concept Analysis*, Darmstadt, 2009. Verlag Allgemeine Wissenschaft.
9. A. Sestier. Contributions à une théorie ensembliste des classifications linguistiques. (Contributions to a set–theoretical theory of classifications). In *Actes du Ier Congrès de l'AFCAL*, pages 293–305, Grenoble, 1960.
10. Christian Wurm. Completeness of Full Lambek calculus for syntactic concept lattices. In *Proceedings of the 17th Conference on Formal Grammar, Springer Lecture Notes in Computer Science*, in press.