Applicative Abstract Categorial Grammars

Oleg Kiselyov

Tohoku University, Japan
oleg@okmij.org

Abstract

We present the grammar/semantic formalism of Applicative Abstract Categorial Grammars (AACG), based on the recent techniques from functional programming: applicative functors, staged languages and typed final language embeddings. AACG is a generalization of Abstract Categorial Grammars (ACG), retaining the benefits of ACG as a grammar formalism and making it possible and convenient to express a variety of semantic theories.

We use the AACG formalism to uniformly formulate Potts' analyses of expressives, the dynamic-logic account of anaphora, and the continuation tower treatment of quantifier strength, quantifier ambiguity and scope islands. Carrying out these analyses in ACG required compromises with the accompanying ballooning of parsing complexity, or was not possible at all. The AACG formalism brings modularity, which comes from the compositionality of applicative functors, in contrast to monads, and the extensibility of the typed final embedding. The separately developed analyses of expressives and QNP are used as they are to compute truth conditions of sentences with both these features.

AACG is implemented as a 'semantic calculator', which is the ordinary Haskell interpreter. The calculator lets us interactively write grammar derivations in a linguist-readable form and see their yields, inferred types and computed truth conditions. We easily extend fragments with more lexical items and operators, and experiment with different semantic-mapping assemblies. The mechanization lets a semanticist test more and more complex examples, making empirical tests of a semantic theory more extensive, organized and systematic.

1 Introduction

Abstract Categorial Grammars (ACG) [3] are the insightful grammar formalism directly based on (linear) typed lambda-calculus. It elegantly expresses CFG and TAG grammars, lends itself to efficient parsing and lets us uniformly investigate string, tree, and other languages [5, §2,§3]. ACG may also represent syntax-semantic interface, by regarding the surface form of a sentence and the logical formula denoting its meaning as two different interpretations of an abstract form [8], [5, §4]. All three forms are represented in ACG uniformly, as closed (normal) terms in a typed linear lambda-calculus.

However, ACG as a framework to express and analyze grammars and build efficient parsers is not suitable for semantic analyses. Although linear lambda-calculus is necessary for the grammar formalism, it is an obstacle in semantics: for example, the meaning of an intersecting adjective such as 'brown' is expressed as a non-linear term $\lambda P.\, \lambda x.\, \mathbf{brown}\ x \wedge P\ x$. Similar terms arise in the analyses of QNP. Therefore, for the sake of semantic analyses non-linear terms have to be added [5, §4], which destroys the ACG uniformity and elegance and complicates formal investigations. Further, the published analyzes of QNP relied on the abstract form with lifted types, which drastically, algorithmically increases the complexity of parsing from the surface to the abstract form. Thus a form of ACG suitable for syntactic analyses is not suitable for semantics, and vice versa. Furthermore, phenomena like scope islands proved problematic for ACG.

We present the generalization of ACG, called Applicative ACG (AACG), that overcomes the painful trade-offs and limitations. AACGs are based on the techniques recently developed in functional programming. First, we apply staging [7] to cleanly separate the (object) language in which to write the abstract, logic, and other forms – and the (meta)language to compute these forms. To transform languages (e.g., map the abstract form to the logic formula) we rely on the typed final, "tagless-final" interpretations [1] as a framework for embedding one language in another. Like ACG, the tagless-final approach is compositional and encourages multiple interpretations of the same object language; unlike ACG, not only base types but also connectives (arrows) may be given different meaning in different interpretations; they may be treated as lollipops in one interpretation and regular arrows in another.

Lastly, we use computational effects to express in a modular fashion the interaction of a phrase with its context, which underlies scope-taking, anaphora, etc. Type lifting is one particular way of expressing effects; there are others, more general and extensible. Effects tend to evoke in a functional programmer (and recently, in a linguist [2]) "monads". We demonstrate that for linguistic analyses, more useful is a different way of expressing effects, so-called applicative functors, see §2.2. They are grounded in Category Theory (even more so than what passes for monads in functional programming), and, unlike monads, composable.

AACG keeps the advantages of ACG as the grammar formalism and, at the same time and uncompromisingly, also lets us write a variety of semantic analyses, *including those not possible before*. §5 compares ACG with AACG in more detail. Here we point out the main methodological difference, coming from staging. In a departure from the Categorial Grammar tradition, the rules of grammatical and semantic composition are specified by the evaluation rules of the metalanguage – and only incompletely by the underlying type calculus. Types are quickly-decidable approximation of the evaluation rules. Whereas type-logical grammars stress finding proofs, we stress deterministic evaluation. Indeed, as Moschovakis said, "the sense of an expression is the algorithm that allows one to compute the denotation of the expression."

The next section presents the major components of AACG: languages to write the forms and denotations and the language to compute them; applicative functors; and the lexicon, the language transformer. §3 illustrates that the surface and the abstract forms are related in AACG essentially in the same way as they are in ACG. §4 shows off AACG as a semantic framework: in §4.1 we write Potts' analyses of expressives, and in §4.2 of QNP without type raising. To illustrate the modularity and the compositionality, §4.3 treats phrases with expressives and QNP, reusing the two previous analyses *as they are*.

We have implemented AACG in Haskell; the complete code is available at `http://okmij.org/ftp/gengo/applicative-symantics/`. We can interactively compute AACG yields and denotations, extend fragments, add new analyses and combine the old ones. The mechanization lets us test more and more complex examples. In particular, the code contains the analyses of scope islands, not possible previously with the original ACG.

2 Applicative Abstract Categorial Grammars

This section briefly and fairly formally presents AACG. We start with T-languages: term languages in which to write the surface (pheno) form of sentences, abstract form(s) and the semantic form. We then describe lambda-calculus with T-language constants. §2.2 introduces applicative functors. Finally we present the lexicon, the mapping between T-languages, which serves, among others, as the syntax-semantic interface.

Base types		υ
T-Types	σ ::=	$\upsilon \mid \sigma \rightarrowtail \sigma$
T-Constants		c
T-Terms	d ::=	$c \mid d \mathbin{\scriptstyle\blacksquare} d$

$$\text{Typing rules}\qquad \frac{}{c : \sigma} \qquad \frac{d_1 : \sigma_1 \rightarrowtail \sigma_2 \quad d_2 : \sigma_1}{d_1 \mathbin{\scriptstyle\blacksquare} d_2 : \sigma_2}$$

Figure 1: A T-language

2.1 Term Languages and the Calculus to Write them

Figure 1 defines a T-language, the analogue of the higher-order signature of ACG. T-types σ are formed from base types υ and the binary connective - $\rightarrowtail$ -. T-terms d are formed from constants c and the binary connective -$\mathbin{\scriptstyle\blacksquare}$-. Each constant is assigned a T-type. A T-language is the set of all well-typed terms – or, a set of finite trees. On the latter view, the constants with their assigned types constitute a tree grammar. Different T-languages differ in their set of base types and their constants.

The following table shows three sample T-languages to be used throughout. The table lists

	T_A	T_S	T_L
Base types	S, NP, N, VP = NP $\rightarrowtail$ S	string	e, t
Constants	John : NP donkey : N saw : NP $\rightarrowtail$ VP every, a, the : N $\rightarrowtail$ NP brown : N $\rightarrowtail$ N	$\cdot$: string $\rightarrowtail$ string $\rightarrowtail$ string `"John"`, `"donkey"`, `"every"`, `"a"`, ...	**john** : e **donkey** : $e \rightarrowtail t$ **see** : $e \rightarrowtail e \rightarrowtail t$ $\wedge : t \rightarrowtail t \rightarrowtail t$ $\forall, \exists : (e \rightarrowtail t) \rightarrowtail t$ $x, y, z, \ldots : e$ $\hat{x} : t \rightarrowtail (e \rightarrowtail t)$

only representative constants; we will tacitly add similar constants (e.g., man : N) as needed. The language T_S, with the single base type string, expresses the surface form of a sentence. It has a multitude of string constants like `"John"` plus the function constant -$\cdot$-. We will write the composite T_S terms like

(-$\cdot$-)$\mathbin{\scriptstyle\blacksquare}$`"John"`$\mathbin{\scriptstyle\blacksquare}$((-$\cdot$-)$\mathbin{\scriptstyle\blacksquare}$`"saw"`$\mathbin{\scriptstyle\blacksquare}$((-$\cdot$-)$\mathbin{\scriptstyle\blacksquare}$`"a"`$\mathbin{\scriptstyle\blacksquare}$`"donkey"`))

simply as

`"John"` $\cdot$ `"saw"` $\cdot$ `"a"` $\cdot$ `"donkey"`

effectively treating -$\cdot$- as the binary infix operator denoting string concatenation.

In the language T_A we write the abstract representation of the sentence. It has the familiar categorial base types S, NP and N; the type VP is an abbreviation for $NP \rightarrowtail S$. T_L is the language of the first-order logic, to express the meaning of a sentence. It has domain constants such as **john** and logical constants like $\wedge$ (which we also treat as the binary infix operator and overload for other types such as $(e \rightarrowtail t) \rightarrowtail (e \rightarrowtail t) \rightarrowtail (e \rightarrowtail t)$). There is the countable number of constants x, y, z, etc., of the base type e, and the corresponding constants $\hat{x} : t \rightarrowtail (e \rightarrowtail t)$. Hence T_L, unlike T_S, effectively has abstraction, notated as $\hat{x}\mathbin{\scriptstyle\blacksquare}d$. The abstraction is introduced

Base types		σ
Types	τ ::=	$\sigma \mid \tau \to \tau$
Constants		d
Terms	e ::=	$d \mid e\ e \mid x \mid \lambda x.\, e$

Figure 2: A $\Lambda(T)$-language

only when needed, rather than universally as in ACG. (With ACG, one has to work hard to avoid the unwanted abstraction,[5, §3].)

Figure 2 defines the language of lambda-terms over a T-language. It is the standard simply-typed lambda-calculus whose base types are the T-types σ and whose constants are T-terms. The typing rules are standard and elided. We write as $\Lambda^{\multimap}(T)$ a linear-typed subset of $\Lambda(T)$.

Lambda-calculus has the standard notions of substitution, β- and η-reductions and normalization. Although T_L looks like lambda-calculus, it, unlike $\Lambda(T)$, has no notion of reduction or substitution, aside from meta-theory. T-languages are mere specifications whereas $\Lambda(T)$ and applicative functors below are the facilities to build these specifications. This clear (phase) separation between specification and computing it is one of the salient distinctions between AACG and ACG.

2.2 Applicative Functors

Let $\mathcal{F}$ be a functor in the Category-Theory sense, the mapping of types and terms, with the property $\mathcal{F}[\tau_1 \to \tau_2]$ is $\mathcal{F}[\tau_1] \to \mathcal{F}[\tau_2]$ and for each term $e : \tau_1 \to \tau_2$ we have $\mathcal{F}[e] : \mathcal{F}[\tau_1] \to \mathcal{F}[\tau_2]$. In the following we apply $\mathcal{F}$ to types only. The action of the functor on terms is written as applications of the following constants (- $\star$ - is taken to be the infix operator). In this

$$\eta : \tau \to \mathcal{F}[\tau] \qquad \mathsf{map} : (\tau_1 \to \tau_2) \to \mathcal{F}[\tau_1] \to \mathcal{F}[\tau_2]$$
$$\text{-} \star \text{-} : \mathcal{F}[\tau_1 \to \tau_2] \to \mathcal{F}[\tau_1] \to \mathcal{F}[\tau_2] \qquad \Downarrow : \mathcal{F}[\tau] \to \tau'$$

explicit form $\mathcal{F}$ is what is known in functional programming as *applicative functor* (applicative, for short) [6]. The explicitness, sometimes necessary to avoid confusion, leads to boilerplate, which we eliminate with the convenient notation of idiomatic brackets introduced in [6]. If $f : \tau_1 \to \tau_2 \to \tau$, $x : \tau_1$ and $y : \mathcal{F}[\tau_2]$, then the idiomatic expression $[\![f\ x\ y]\!]$ means $\eta f \star \eta x \star y$, or, alternatively, $\mathsf{map}\ (f\ x)\ y$.

The above table also shows the operation $\Downarrow$, which is unusual in that it is generally *partial*: it may undefined for some terms. The result type of $\Downarrow$, although generally related to τ, may differ from it, depending on the particular applicative. We will see the examples later.

Functors compose: whenever $\mathcal{F}_1$ and $\mathcal{F}_2$ are functors, their composition $\mathcal{F}_1 \circ \mathcal{F}_2$ is a functor as well. This is in marked contrast with monads, which do not compose.

The identity functor $\mathcal{F}_{id}$ is the identity mapping. A sample non-trivial applicative is partiality $\mathcal{F}_m$. To define it, we add unit () and the sum type $\tau_1 \oplus \tau_2$ in the standard way, with introduction constants $\mathsf{inl} : \tau_1 \to \tau_1 \oplus \tau_2$, $\mathsf{inr} : \tau_2 \to \tau_1 \oplus \tau_2$ and the elimination form $\mathsf{case}\, e\ \{\mathsf{inl}\, x \to e_1 \mid \mathsf{inr}\, y \to e_2\}$. The morphisms of the functor are defined as follows (eliding the straightforward map). We call $\mathsf{inl}()$ a failure, which propagates on further mappings. The operation $\Downarrow$ is indeed partial, being undefined on failure.

$$\mathcal{F}[\tau] = () \oplus \tau$$
$$\eta = \mathsf{inr} \qquad \mathsf{fail} = \mathsf{inl}()$$
$$\text{-} \star \text{-} = \lambda uv.\, \mathsf{case}\ u\ \{\mathsf{inl}\ x \to \mathsf{fail}\ \mid\ \mathsf{inr}\ z \to \mathsf{case}\ v\ \{\mathsf{inl}\ x \to \mathsf{fail}\ \mid\ \mathsf{inr}\ y \to \mathsf{inr}\ z\ y\}\}$$
$$\Downarrow = \lambda u.\, \mathsf{case}\ u\ \{\mathsf{inl}\ x \to \bot\ \mid\ \mathsf{inr}\ x \to x\}$$

2.3 Lexicon: the T-language-to-language Interface

An AACG lexicon $\mathcal{L}$ (named after ACG lexicons) is a mapping from one T-language T^1 (with base types υ^1 and constants c^1) to another T-language T^2, determined as follows. First, expressions of T^1 are mapped to expressions of $\Lambda(\mathcal{F}[T^2])$ (where the choice of $\mathcal{F}$ depends on the lexicon). This mapping, which we write as $\mathcal{L}[d]$, is homomorphic: a constant $c^1 : \sigma^1$ is mapped to a lambda-term of the type $\mathcal{L}[\sigma^1]$, and an expression $d_1 \centerdot d_2$ to an application: $\mathcal{L}[d_1 \centerdot d_2] = \mathcal{L}[d_1]\ \mathcal{L}[d_2]$. The type mapping is also homomorphic: Base-types of T^1 are mapped to generally non-base types of T^2: $\mathcal{L}[\upsilon^1] = \mathcal{F}[\sigma^2]$. Non-based types of T^1 are mapped homomorphically: $\mathcal{L}[\sigma_1 \rightarrowtail \sigma_2] = \mathcal{L}[\sigma_1] \to \mathcal{L}[\sigma_2]$.

Theorem 1. *For all d^1 of T^1, $\mathcal{L}[d^1]$ is well-typed in $\Lambda(\mathcal{F}[T^2])$.*

Theorem 2. *For all $d^1 : \upsilon^1$ of a base type of T^1, $\mathcal{L}[d^1]$ has the normal form of the type $\mathcal{F}[\sigma^2]$ for some σ^2.*

To stress, the lexicon maps a T-term d^1 to a lambda-term, which most likely will have redices and has to be normalized. If the normal form is of the type $\mathcal{F}[\sigma]$ for some σ, $\Downarrow$ may be applied, which, if defined, produces a T-term – taken to be the lexicon-mapping result of the original T-term. We will see many examples of such transformations later. For now we observe that the lexicon mapping is inherently partial. The failure of the mapping indicates some form of ungrammaticality (inadmissibility) of the sentence represented by the original T-term. We also see the interplay of compositionality and context-sensitivity: the mapping $\mathcal{L}[d]$, being homomorphism, is compositional. The normalization of the term and extracting the result may not be (that depends on the functor in question). We see the examples next.

3 AACG as a Grammar Formalism

This section demonstrates the mapping between the abstract and the surface forms: the syntactic side of AACG, as a grammar formalism. We shall see that the mapping is established in essentially the same way as in ACG, meaning that the parsing results and techniques developed for ACG also apply to AACG.

Recall, the abstract form is a T_A-term whereas the surface form is represented by a T_S (string) term. The lexicon that maps from T_A to T_S will use as an example the following T_A term:

$$\mathsf{saw} \centerdot (\mathsf{the} \centerdot (\mathsf{brown} \centerdot \mathsf{donkey})) \centerdot \mathsf{john} : \mathsf{S}$$

The lexicon $\mathcal{L}_{syn}$, Fig 3 maps the constants of T_A to linear lambda-terms $\Lambda^{\multimap}(\mathcal{F}[T_S])$. All base types of T_A are mapped to $\mathcal{F}[\mathsf{string}]$. The base-type constants john and donkey are mapped to the corresponding strings, injected into the applicative. Since brown is not of a base type it becomes an $\mathcal{F}[\mathsf{string}] \to \mathcal{F}[\mathsf{string}]$ function. For our sample term, we have

$$\begin{array}{ll}
\mathsf{john} : \mathsf{NP} & \mapsto \eta\text{"}John\text{"} \\
\mathsf{donkey} : \mathsf{N} & \mapsto \eta\text{"}donkey\text{"} \\
\mathsf{brown} : \mathsf{N} \rightarrowtail \mathsf{N} & \mapsto \lambda x. [\![\text{"}brown\text{"} \cdot\ x]\!] \\
\mathsf{the} : \mathsf{N} \rightarrowtail \mathsf{NP} & \mapsto \lambda x. [\![\text{"}the\text{"} \cdot\ x]\!] \\
\mathsf{saw} : \mathsf{NP} \rightarrowtail \mathsf{NP} \rightarrowtail \mathsf{S} & \mapsto \lambda x_o x_s. [\![x_s \cdot\ (\text{"}saw\text{"} \cdot\ x_o)]\!]
\end{array}$$

Figure 3: $\mathcal{L}_{syn}$: mapping to the surface string

$$\begin{array}{ll}
 & \mathcal{L}[\mathsf{saw} \,\blacksquare\, (\mathsf{the} \,\blacksquare\, (\mathsf{brown} \,\blacksquare\, \mathsf{donkey})) \,\blacksquare\, \mathsf{john}] \\
 & \{\text{lexicon homomorphism}\} \\
= & \mathcal{L}[\mathsf{saw}]\ (\mathcal{L}[\mathsf{the}]\ (\mathcal{L}[\mathsf{brown}]\ \mathcal{L}[\mathsf{donkey}]))\ \mathcal{L}[\mathsf{john}] \\
 & \{\text{substituting the } \mathcal{L}_{syn} \text{ mapping of constants}\} \\
= & (\lambda x_o x_s. [\![x_s \cdot\ (\text{"}saw\text{"} \cdot\ x_o)]\!])\ ((\lambda x. [\![\text{"}the\text{"} \cdot\ x]\!])\ ((\lambda x. [\![\text{"}brown\text{"} \cdot\ x]\!])\ \eta\text{"}donkey\text{"}))\ \eta\text{"}John\text{"} \\
 & \{\text{normalizing}\} \\
= & [\![\text{"}John\text{"} \cdot \text{"}saw\text{"} \cdot \text{"}the\text{"} \cdot \text{"}brown\text{"} \cdot \text{"}donkey\text{"}]\!] \\
 & \{\text{desugaring idiom brackets}\} \\
= & \eta(\text{"}John\text{"} \cdot \text{"}saw\text{"} \cdot \text{"}the\text{"} \cdot \text{"}brown\text{"} \cdot \text{"}donkey\text{"}) : \mathcal{F}[\mathsf{string}]
\end{array}$$

Choosing $\mathcal{F}_{id}$ for the applicative $\mathcal{F}$ and applying its $\Downarrow$ (which is the identity) yields the surface form of the sentence. Recall, the intended meaning of $- \cdot -$ in T_s is string concatenation.

Our derivation of the surface form is identical to that of ACG, except for the applicative – which is the identity functor anyway. One may envision more interesting applicatives, such as $\mathcal{F}[\tau] = (\tau, \{\mathsf{attribute}\})$ pairing with each type a set of attributes (case, gender, number, etc). The AACG formalism then expresses Minimalist grammars (the operation $- \star -$ becomes the Merge).

Like the ACG derivation, ours uses only linear lambda-terms. In ACG, the abstract form is also written in the linear lambda calculus; our T_A is vacuously linear as it has no abstractions (which is enough for modeling TAGs, for example, [5, §3]). Abstractions are easy to add (see T_L for example), at which point we may restrict them to be linear. We may also use a lexicon mapping to general lambda-terms, and enforce linearity differently, through an applicative $\mathcal{F}[\tau]_{lin} = () \oplus (\tau, \{\mathsf{attribute}\})$ where this time $\mathsf{attribute}$ is a unique identifier associated with each constant. The operation $- \star -$ merges the attributes, checking for duplicates. Encountering a duplicate attribute signals failure, as in the partiality applicative $\mathcal{F}_m$ of §2.2. Thus *effectively* $\Lambda(\mathcal{F}[T]_{lin})$ is $\Lambda^{\multimap}(\mathcal{F}[T]_{id})$.

4 Syntax-Semantics Interface

This section describes another mapping of the abstract form T_A, not to a string but to a first-order logic formula – the meaning of the sentence represented by the original T_A term. This mapping, together with the one in §3, (indirectly) relates syntax and semantics. The abstract language can be transformed to several concrete languages; there may also be alternative semantics transformations, which hence account for ambiguity.

This section highlights the modularity of AACG: first we set up the lexicon to obtain the meaning of simple sentences; next we add expressives to our fragment to reproduce Potts' analyses, re-using the vanilla lexicon. §4.2 extends the base fragment in a different way, with indefinites. Finally we analyze the sentences with both expressives and indefinites, re-*using* rather than re-*doing* the base lexicon and its two separate extensions. Such modularity is a distinct feature of AACG: ACG has to use type-lifting for QNP and hence it cannot reuse the unlifted fragment.

We start with the sample T_A term from §3, but transform it to a logic formula (T_L term) rather than the T_S string, using the following lexicon $\mathcal{L}_{sym}^{basic}$. The lexicon is a more elaborate

$\mathsf{john} : \mathsf{NP}$	$\mapsto \eta\mathbf{john}$	$: \mathcal{F}[e]$
$\mathsf{donkey} : \mathsf{N}$	$\mapsto \eta\mathbf{donkey}$	$: \mathcal{F}[e \rightarrowtail t]$
$\mathsf{brown} : \mathsf{N} \rightarrowtail \mathsf{N}$	$\mapsto \lambda x.\, [\![\mathbf{brown} \wedge x]\!]$	$: \mathcal{F}[e \rightarrowtail t] \to \mathcal{F}[e \rightarrowtail t]$
$\mathsf{the} : \mathsf{N} \rightarrowtail \mathsf{NP}$	$\mapsto \lambda x.\, [\![\iota \bullet x]\!]$	$: \mathcal{F}[e \rightarrowtail t] \to \mathcal{F}[e]$
$\mathsf{saw} : \mathsf{NP} \rightarrowtail \mathsf{NP} \rightarrowtail \mathsf{S}$	$\mapsto \lambda x_o x_s.\, [\![\mathbf{see} \bullet x_o \bullet x_s]\!]$	$: \mathcal{F}[e] \to \mathcal{F}[e] \to \mathcal{F}[t]$

version of $\mathcal{L}_{syn}$ from §3, built on the same principle: constants of the base types like john and donkey are mapped to T_L constants injected into an applicative. The other constants are mapped to $\Lambda(\mathcal{F}[T_L])$ functions.

We have specified $\mathcal{L}_{sym}^{basic}$ without choosing the applicative $\mathcal{F}$: we did not have to, since it is valid in any applicative. Choosing $\mathcal{F}_{id}$ gives us, in the same process we saw in §3, the following T_L formula for the running example

$$\mathbf{see} \bullet\ (\iota \bullet (\mathbf{brown} \wedge \mathbf{donkey})) \bullet \mathbf{john}$$

4.1 Expressives

We now demonstrate how to faithfully represent in the AACG framework the Potts analysis [9] of expressives. Our running example will be the T_A term

$$\mathsf{saw} \bullet (\mathsf{the} \bullet (\mathsf{damn} \bullet \mathsf{donkey})) \bullet \mathsf{john} : \mathsf{S}$$

It includes the sample expressive constant $\mathsf{damn} : \mathsf{N} \rightarrowtail \mathsf{N}$, with the same type as brown and behaving syntactically as an ordinary adjective; the mapping to the surface form is exactly as in §3.

Potts regards the at-issue content and the expressive content as two separate, non-interacting dimensions of meaning. We accumulate the latter in an applicative. We introduce a small T-language T_E for the expressive content, with the base type a and the constants $\mathsf{nip} : a$, $\mathsf{annoy} : a$ and $\cdot : a \rightarrowtail a \rightarrowtail a$, to model the list of attitudes. The expressive applicative $\mathcal{F}_{exp}$ is defined below (in the lambda-calculus with pairs). That is, $\mathcal{F}_{exp}$ pairs a term with the expressive

$$\mathcal{F}[\tau] = (\tau, a) \quad \eta = \lambda x.\, (x, \mathsf{nip}) \quad \Downarrow = \lambda x.\, x$$
$$\text{-} \star \text{-} = \lambda uv.\, ((\mathsf{fst}\ u)(\mathsf{fst}\ v), (\mathsf{snd}\ u) \cdot (\mathsf{snd}\ v))$$

content, which is accumulated on each operation. To obtain the truth conditions for the sample sentence, we only need to specify the semantic mapping for damn

$$\mathsf{damn} : \mathsf{N} \rightarrowtail \mathsf{N} \quad \mapsto \quad \lambda x.\, (\mathsf{fst}\ x, \mathsf{annoy} \cdot (\mathsf{snd}\ x)) : \mathcal{F}[e \rightarrowtail t] \to \mathcal{F}[e \rightarrowtail t]$$

which keeps the at-issue content and adds the annoy attitude to the expressive content. Recall that $\mathcal{L}_{sym}^{basic}$ was specified for any applicative. Therefore, we can reuse it *as it is*, obtaining the expected denotation for our sample sentence, the (T_L, T_E) pair

$$(\mathbf{see} \bullet\ (\iota \bullet \mathbf{donkey}) \bullet \mathbf{john}, \mathsf{annoy} \cdot \mathsf{nip})$$

A more elaborate example

$$\mathsf{in} \bullet (\mathsf{the} \bullet (\mathsf{damn} \bullet \mathsf{field}))\ (\mathsf{saw} \bullet (\mathsf{the} \bullet (\mathsf{damn} \bullet (\mathsf{brown} \bullet \mathsf{donkey})))) \bullet \mathsf{john} : \mathsf{S}$$

gets the following meaning

$$(\mathsf{in} \bullet\ (\iota \bullet \mathsf{field})\ (\hat{x}\ (\mathbf{see} \bullet\ (\iota \bullet (\mathbf{brown} \wedge \mathbf{donkey})) \bullet x)) \bullet \mathbf{john}, \mathsf{annoy} \cdot \mathsf{annoy} \cdot \mathsf{nip})$$

The AACG representation of Potts analysis faithfully reproduces its salient features: the -$\star$- operation of $\mathcal{F}_{exp}$ makes the expressive content a simple sum of contributions from all parts of the sentence; the reuse of $\mathcal{L}_{sym}^{basic}$, defined for any applicative, ensures that the basic lexical items do not affect the expressive content. By design, the side-effect of an applicative cannot depend on the value it produces. Using the applicative hence guarantees that the content at issue cannot affect the expressive dimension.

4.2 Quantification

Another way of extending the basic fragment is adding quantifiers, such as the new T_A constant $\mathsf{a} : \mathsf{N} \rightarrowtail \mathsf{NP}$. It has the same type as the and syntactically behaves as a regular determiner. To

$$\begin{array}{l}\mathcal{F}[\tau]_{CPS} = (\tau \to \mathcal{F}[t]) \to \mathcal{F}[t] \\ \eta = \lambda x.\, \lambda k.\, k\ x \qquad \Downarrow = \lambda x.\, x\ (\lambda z.\, z) \\ \text{-}\star\text{-} = \lambda uv.\, \lambda k.\, u\ (\lambda x.\, v\ (\lambda y.\, k\ (x\ y)))\end{array}$$

compute its semantics we introduce the applicative $\mathcal{F}_{CPS}$ indexed by some other applicative $\mathcal{F}$, and define the lexicon mapping for the indefinite thusly

$$\mathsf{a} : \mathsf{N} \rightarrowtail \mathsf{NP} \quad \mapsto \quad \lambda u.\, \lambda k.\, u\ (\lambda x.\, [\![\exists \centerdot (\hat{z} \centerdot (x \centerdot z) \wedge (k\ z)]\!] : \mathcal{F}[e \rightarrowtail t]_{CPS} \to \mathcal{F}[e]_{CPS}$$

That is the only addition to be able to compute the meaning of the sentences such as (in the abstract form) saw . (a . (brown . donkey)) . john : S. For constants other than the indefinite we reuse $\mathcal{L}^{basic}_{sym}$, taking its applicative to be $\mathcal{F}_{CPS}$.

The applicative $\mathcal{F}_{CPS}$ is defined in terms of the continuation passing and type-lifting – which is hidden in the $\mathcal{F}_{CPS}$ mapping. Emphatically, the abstract term T_A did not have lifted types. That is why we could re-use the basic lexicon for the quantifier-free fragments. Such AACG modularity comes from mapping of arrow types, and hence is not possible in ACG.

4.3 Expressives and QNP

Finally we combine the previous analyses to compute the meaning of the phrase with both expressives and indefinites, represented by the T_A term

saw . (a . (damn . donkey)) . john : S

There is nothing to more to define: we use the applicative that is the composition of $\mathcal{F}_{exp}$ and $\mathcal{F}_{CPS}$. Such composition is not possible with monads.

The file `Sem.hs` in the accompanying code has the complete development. The code highlights interactivity, progressively extending the fragments with more and more features, and type inference.

5 AACG and ACG

AACG is a more expressive reformulation of ACG that makes it possible to carry out semantic analyses without compromising syntactic ones. The AACG lexicon interprets not only the terms and base types but also arrows. Therefore, syntactic analyses may be formulated entirely in the linear lambda-calculus, as in the original ACG, whereas semantic ones may use non-linear terms. Lifted types that appear in some semantic analyses do not "leak" into the abstract signature; the latter may remain second order, ensuring the efficient parsing. AACG is thus more modular: for sentences with both expressives and QNP we used two separately developed analyses, one of which relied on the lifted types and the other did not. Finally, AACG lets us analyze scope islands, which were thought problematic with ACG.

AACG and ACG share the same intuition, coming from Curry's tectogrammatics. The surface and logical forms – syntax and semantics – are obtained by transforming the common abstract form. The idea of multiple, composable interpretations unites ACG and AACG. What separates AACG is staging, distinguishing the object language (to write the forms in) from the metalanguage to build them. Whereas the metalanguage needs the full power of lambda-calculus, the object language does not have to[1]. (In fact, the abstraction can be undesirable, see [5, §4] for examples.) Even when an object language does have abstractions (as T_L for example), there is no, aside from meta-theory, corresponding notion of substitution or reduction. T-expressions are not meant to be evaluated or normalized, they are frozen specifications. $\Lambda(T)$ and applicative functors are the facilities to build these specifications.

As in ACG, a T-language transformed by a lexicon may be further transformed by another lexicon: the transformations compose, and there may be several abstract languages, with various degrees of "abstractness", transformed in several steps to the surface or logical forms.

We stress that on the syntactic side, relating the surface form of a sentence with the abstract form, AACG is essentially the same as ACG. Therefore, all results of ACG parsing immediately apply to AACG.

6 Related Work

AACG shares the overall goals and aspirations of Convergent Grammars (CVG) [4]: both have the parallel architecture, are weakly syntacticocentric and uncannily resemble the transformational grammar approaches of the 70s[2]. CVG sets up the syntax-semantic interface by recursively pairing syntactic and semantic derivations. In AACG, as ACG, the pairing is implicit in the derivation of syntactic and semantic forms from the common abstract form. Either way, the syntax-semantic interface is not a function – even more so in AACG because the derivations are not total (see the partiality applicative). CVG relies on Gazdar and Cooper-storage ideas to model the context sensitivity (i.e., apparent non-compositionality) of meaning of some expressions such as pronouns or quantifiers. AACG borrows instead from Computer Science the concept of applicatives, with its clear foundations and well-investigated properties and equational laws.

We have called ACG and AACG a semantic and grammar framework because it lets us write (abstract) grammars of various language fragments and produce their yields and semantic derivations. The semantic and grammar framework, Grammatical Framework (GF) [10] immediately springs to mind. First of all, AACG and GF – like SciPy and Matlab for example – are in polar weight categories. AACG is a small *embedded* DSL for syntactic and semantic analyses: a small, easily modifiable Haskell library. GF is a full-blown programming language oriented towards tree manipulation, with impressively many tools and features for writing grammars for entire natural languages, generating parsers, machine translation, etc. GF has decades of development, many participants, large resources and now a commercial company.

The second, methodological difference is that GF is a general-purpose grammatical framework: it lets one write CCG analyses, or ACG, TLG or even HPSG. It has excellent facilities for manipulating typed trees. On the other hand, AACG (like ACG or CCG) is meant to be a linguistic theory – a particular methodology of relating the surface form of a sentence with its semantics.

[1]Higher-order facilities can be introduced, in limited form, through combinators

[2]`http://www.coffeeblack.org/cvg/overview.html`

7 Conclusions

We have presented AACG, a grammar and semantic formalism. The idea of staging lets us separate the languages in which to write the surface, meaning and other formulas – and in which to build them. The transformations from the abstract form to a concrete (meaning or surface form) uses both the compositional, functorial mapping and the evaluation in an applicative, which models the apparent non-compositionality of interaction with context. As Moschovakis posited, the sense of an expression is indeed the algorithm that allows one to compute the denotation of the expression.

We have implemented AACG as a domain-specific language embedded in Haskell, taking the full advantage of the abstraction, modularity, and interactive development and debugging facilities of the host language.

The immediate future work is analyzing non-canonical coordination and gapping.

This work was partially supported by JSPS KAKENHI Grant Numbers 22300005, 25540001, 15H02681.

References

[1] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[2] Simon Charlow. *On the semantics of exceptional scope*. PhD thesis, New York University, USA, 2014.

[3] Philippe de Groote. Towards abstract categorial grammars. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 148–155, San Francisco, CA, July 2002. Morgan Kaufmann.

[4] Philippe de Groote, Sylvain Pogodalla, and Carl Pollard. On the syntax-semantics interface: From convergent grammar to abstract categorial grammar. In Hiroakira Ono, Makoto Kanazawa, and Ruy de Queiroz, editors, *Logic, Language, Information and Computation*, volume 5514 of *Lecture Notes in Computer Science*, pages 182–196. Springer Berlin Heidelberg, 2009.

[5] Makoto Kanazawa and Sylvain Pogodalla. Advances in Abstract Categorial Grammars: Language theory and linguistic modeling. Lecture notes, ESSLLI 09. Part 2, July 2009.

[6] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, January 2008.

[7] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, 1992.

[8] Sylvain Pogodalla. Generalizing a proof-theoretic account of scope ambiguity. In *IWCS-7*, 2007.

[9] Christopher Potts. The expressive dimension. *Theoretical Linguistics*, 33:165–198, 2007.

[10] Aarne Ranta. Grammatical Framework: A type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004.