A constrained graph algebra for semantic parsing with AMRs

Jonas Groschwitz[*†] Meaghan Fowlie[*] Mark Johnson[†] Alexander Koller[*]
[*] Saarland University, Saarbrücken, Germany [†] Macquarie University, Sydney, Australia
jonasg|mfowlie|koller@coli.uni-saarland.de
mark.johnson@mq.edu.au

Abstract

When learning grammars that map from sentences to abstract meaning representations (AMRs), one faces the challenge that an AMR can be described in a huge number of different ways using traditional graph algebras. We introduce a new algebra for building graphs from smaller parts, using linguistically motivated operations for combining a head with a complement or a modifier. Using this algebra, we can reduce the number of analyses per AMR graph dramatically; at the same time, we show that challenging linguistic constructions can still be handled correctly.

1 Introduction

Semantic parsers are systems which map natural-language expressions to formal semantic representations, in a way that is learned from data. Much research on semantic parsing has focused on mapping sentences to Abstract Meaning Representations (AMRs), graphs which represent the predicate-argument structure of the sentences. Such work builds upon the AMRBank (Banarescu et al., 2013), a corpus in which each sentence has been manually annotated with an AMR.

The training instances in the AMRBank are annotated only with the AMRs themselves, not with the structure of a compositional derivation of the AMR. This poses a challenge for semantic parsing, especially for approaches which induce a grammar from the data and thus must make this compositional structure explicit in order to learn rules (Jones et al., 2012, 2013; Artzi et al., 2015; Peng et al., 2015). In general, the number of ways in which an AMR graph can be built from its atomic parts, e.g. using the generic graph-combining operations of the HR algebra (Courcelle, 1993), is huge (Groschwitz et al., 2015). This makes grammar induction computationally expensive and undermines its ability to discover grammatical structures that can be shared across multiple instances. Existing approaches therefore resort to heuristics that constrain the space of possible analyses, often with limited regard to the linguistic reality of these heuristics.

We propose a novel method to generate a constrained set of derivations directly from an AMR, but without losing linguistically significant phenomena and parses. To this end we present an *apply-modify (AM) graph algebra* for combining graphs using operations that reflect the way linguistic predicates combine with complements and adjuncts. By equipping graphs with annotations that encode argument sharing, AM algebra derivations can model phenomena such as control, raising, and coordination straightforwardly. We describe a method to generate the AM algebra for an AMR in practice, and demonstrate its effectiveness: e.g. for graphs with five nodes, our method reduces the number of candidate terms from 10^{17} to just 21 on average.

The paper is structured as follows. Section 2 reviews some related work and sets the stage for the rest of the paper. Section 3 briefly reviews the HR algebra and its problems for grammar induction, and then defines the AM algebra. In Section 4, we discuss a number of challenging linguistic phenomena and demonstrate that AM algebra derivations can capture the intended compositional derivations of the resulting AMRs. We explain how to obtain AM algebra derivations for graphs in the AMRBank in Section 5. Finally, we show that in practice, we can indeed reduce the set of derivations while achieving high coverage in Section 6.

2 Related work

This paper is concerned with finding the hidden compositional structure of AMRs, the semantic representations annotated in the AMRBank (Banarescu et al., 2013). AMRs are directed, acyclic, rooted graphs with node labels indicating semantic concepts and edge labels indicating semantic roles, such as arguments ARG0, ARG1, ... and modifiers, such as manner and time.

An example is shown in Fig. 1.[1] Here, *snake* is an ARG0 of both *chew* and *swallow*, and *prey* is an ARG1 of *swallow*. Nodes can fill argument positions of multiple predicate nodes, not just because of grammatical phenomena such as control, but also because of coreference (***its** prey*), or because they are pragmatically implied arguments (edge from *chew* to *prey*). An AMR's root represents a "focus" in the graph; the root is often the main predicate of the sentence. In Fig. 1, the root is the *swallow* node. AMRs are "abstract" because they gloss over certain details of the syntactic realization. For example, *destruction of Rome* and *Rome was destroyed* have the same AMR; among others, tense and determiners are dropped.

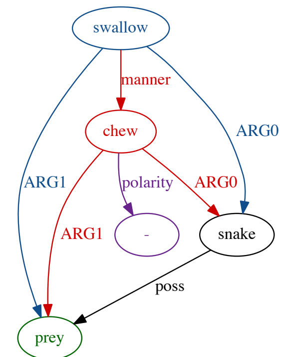


Figure 1: AMR of *The snake swallows its prey without chewing.*

The availability of the AMRBank has spawned much research on semantic parsing into AMR representations. The work in this paper is most obviously connected to research that models the compositional mapping from strings to AMRs with grammars – using either synchronous grammars (Jones et al., 2012; Peng et al., 2015) or CCG (Artzi et al., 2015; Misra and Artzi, 2016). Not all AMR parsers learn explicit grammars (Flanigan et al., 2014; Peng et al., 2017). However, we believe that these, too, may benefit from access to the compositional structure of the AMRs, which the algebra we present makes easier to compute.

The operations our algebra uses to combine semantic representations are closely related to those of the "semantic algebra" of Copestake et al. (2001), which was intended to reflect universal semantic combination operations for large-scale handwritten HPSG grammars. More distantly, the ability of our semantic representations to select the type of its arguments echoes the use of types in Montague Grammar and in CCG (Steedman, 2001), applied to graphs.

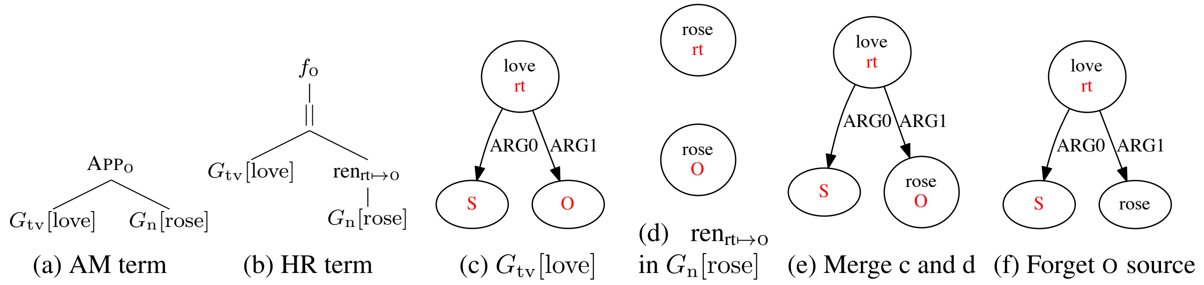
3 Algebras for constructing graphs

We start by reviewing the HR algebra and discussing some of its shortcomings in the context of grammar induction. Then we introduce the apply-modify graph algebra, which tackles these shortcomings in a linguistically adequate way.

Notation: For a given (partial) function $f : A \to B$, we write $\mathcal{D}(f) \subseteq A$ for the set of values on which f is defined and $\mathcal{I}(f) \subseteq B$ for its image. When convenient, we read functions as sets of input-output pairs, so that e.g. $\emptyset$ denotes the partial function that is undefined everywhere. If f, g are (partial) functions, we write $f \circ g$ for the function h such that $h(a) = f(g(a))$ for all a. If f is injective, we write f^{-1} for its inverse (partial) function. We write $\overline{f}$ for the total function such that $\overline{f}(a) = f(a)$ if $f(a)$ is defined and $\overline{f}(a) = a$ otherwise. Finally, for an input value $x \in \mathcal{D}(f)$, we write $f \setminus x$ for the function that is equal to f except that it is undefined on x.

A Σ-*algebra* $\mathbb{A} = \langle \mathcal{A}, (\mathsf{f})_{\mathrm{F} \in \Sigma} \rangle$ is a structure in which terms over a *signature* Σ can be evaluated as elements from the algebra's *domain* $\mathcal{A}$. Here, Σ is a *ranked signature*; that is, a set of symbols $\mathrm{F} \in \Sigma$, each of which is equipped with a *rank* $\in \mathbb{N}$. Symbols of rank 0 are called *constants*. For each $\mathrm{F} \in \Sigma$ of rank k, the algebra defines a function $\mathsf{f} : \mathcal{A}^k \to \mathcal{A}$; in particular, constants are interpreted as elements of $\mathcal{A}$. The functions may be partial; then $\mathbb{A}$ is called a *partial* algebra. We define the *terms* over Σ, T_Σ, recursively: all constants are terms, and if $\mathrm{F} \in \Sigma$ has rank n and $t_1, \ldots, t_n \in T_\Sigma$, then $\mathrm{F}(t_1, \ldots, t_n)$ is

[1] Originally, AMR represents labels on nodes as labelled leaves, i.e. edges of cardinality 1. For readability, we write the labels directly in the nodes, and refer to them as *node labels* in text. We also drop the predicate senses that AMR draws from the OntoNotes project: In reality, e.g. the *swallow* node would be labelled *swallow-01*



(a) AM term (b) HR term (c) $G_{\mathrm{tv}}[\mathrm{love}]$ (d) $\mathrm{ren}_{\mathrm{rt}\mapsto\mathrm{O}}$ in $G_{\mathrm{n}}[\mathrm{rose}]$ (e) Merge c and d (f) Forget O source

Figure 2: HR algebra derivation of *loves a rose*.

also a term. Such a term *evaluates* recursively to the value $[\![t]\!] = \mathsf{f}([\![t_1]\!], \ldots, [\![t_n]\!]) \in \mathcal{A}$. Approaching graphs algebraically allows us to examine the compositionality of graphs – how graphs can be built from smaller graphs. For example, the terms in Figures 2a and 2b both describe the combining of the graphs in Figures 2c and 2d to form the graph in Fig. 2f. We describe the operations used in these terms in the following subsections.

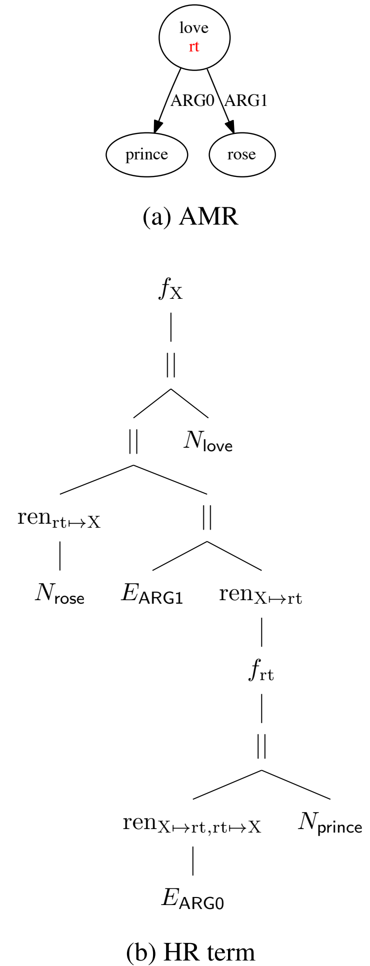
Convention: To increase readability and since in this paper we focus on the functions and rarely refer to a symbol itself, we will denote a function corresponding to a symbol with the symbol itself. I.e. we will use the same notation for both symbol and associated function, not making the distinction between F and f as in the definition above. But as a general principle in this paper, this common notation always refers to the function in text and definitions, and to the symbol in terms such as in Figures 2a and 2b.

3.1 S-graphs and the HR algebra

A standard algebra for the theoretical literature for describing graphs is the *HR algebra* of Courcelle (1993). It is very closely related to hyperedge replacement grammars (Drewes et al., 1997), which have been used extensively for grammars of AMR languages (Chiang et al., 2013; Peng et al., 2015), and Koller (2015) showed explicitly how to do compositional semantic construction using the HR algebra.

The objects of the HR algebra are *s-graphs* $G = (g, S)$, consisting of a graph g (here, directed and with node and edge labels) and a partial function $S : \mathcal{S} \rightsquigarrow V$, which maps *sources* from a fixed finite set $\mathcal{S}$ of source names to nodes of g. Sources thus serve as external, interpretable names for some of the nodes. If we have $S(a) = v$, then we call v an *a-source* of G. An example of an s-graph with a root-source (rt) and a subject-source (S) is shown in Fig. 2f. Sources are marked in diagrams as red node labels.

The HR-algebra serves as a compositional algebra for graphs because it includes an operation *merge* which connects two graphs at the nodes that share source names. The algebra evaluates terms from a signature which, in addition to constants for s-graphs, contains three further types of function symbols. The *merge* operation $||$, of rank two, combines two s-graphs G_1 and G_2 into a new s-graph G' that contains all the nodes and edges of G_1 and G_2. If G_1 has an a-source u and G_2 has an a-source v, for some source name a, then u and v will be mapped to the same node in G', taking with it all the edges into and out of u and v. We will usually write $||$ in infix notation. The *rename* operation $\mathrm{ren}_{\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}}$, of rank one, renames the sources of an s-graph; if the node u was an a_i-source before the rename, it becomes a b_i-source. Finally, the *forget* operation f_a, of rank 1, removes the entry for source a from S, i.e. the resulting s-graph no longer has an a-source.



(a) AMR

(b) HR term

Figure 3: AMR generated by linguistically bizarre HR term

An example for a term of the HR algebra is shown in Fig. 2b. This term uses the constants $G_{\mathrm{tv}}[\mathrm{love}]$ (Fig. 2c) and $G_{\mathrm{n}}[\mathrm{rose}]$, which are evaluated by replacing the node label "$**$" in the graphs G_{tv} and G_{n} in

Fig. 4 with "love" and "rose", respectively. The term *renames* the rt-source of $G_{\mathrm{n}}[\mathrm{rose}]$ to an O-source (Fig. 2d). Thus when the result is *merged* with $G_{\mathsf{tv}}[\mathrm{love}]$, the "rose" label is inserted into the object position of the verb (Fig. 2e). Finally, we *forget* the O-source, yielding the s-graph in Fig. 2f.

The operations of the HR algebra are rather fine-grained, and can be combined flexibly. This is an advantage when developing grammars using the HR algebra by hand (Koller, 2015), but makes the automatic induction of grammars from the AMRBank expensive and error-prone. Groschwitz et al. (2015) show that with more than three sources, the problem becomes quite extreme, and that with such few sources available, one must use constants of only one or two nodes. Following the experimental setup of Groschwitz et al. (2015), N_{rose}, N_{prince} and N_{love} in Figure 3b evaluate to labelled nodes with a rt-source, and E_{ARG0} and E_{ARG1} evaluate to single edges with a rt-source at their source and an X-source at their target. Using these constants and just the two sources rt and X, there are already 3584 terms over the HR algebra which evaluate to the (quite small) s-graph in Fig. 3a.

This set of terms is riddled with spurious ambiguity and linguistically bizarre analyses, such as the term shown in Fig. 3b. Two strange aspects of this example are: one, *prince* becomes an X-source by first switching X and rt and then switching them back; this step is unnecessary and inconsistent with the corresponding process for *rose* here. Two, *prince* and *rose* are combined with empty argument connectors before *love* finally is inserted as the predicate, despite these roles being originally defined in *love*'s semantic frame.

Not only does this make graph parsing computationally expensive (Chiang et al., 2013; Groschwitz et al., 2015), it also makes grammar induction difficult. For example, Bayesian algorithms sample random terms from the AMRBank and attempt to discover grammatical structures that are shared across different training instances. When the number of possible terms is huge, the chance that no two rules share any grammatical structure increases, undermining the grammar induction process. Existing systems therefore apply heuristics to constrain the space of allowable HR terms. However, these heuristics are typically ad-hoc, and not motivated on linguistic grounds. Thus there is a risk that the linguistically correct compositional derivation of an AMR is accidentally excluded.

3.2 The apply-modify graph algebra

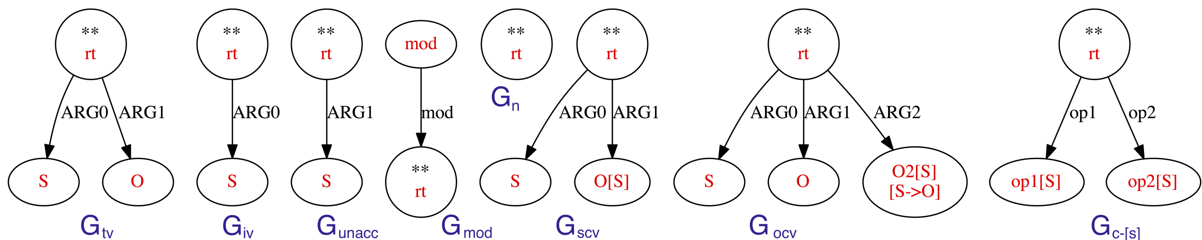
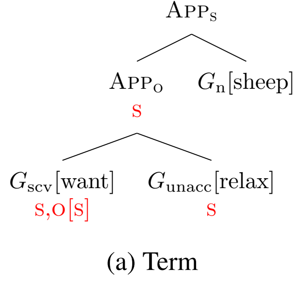
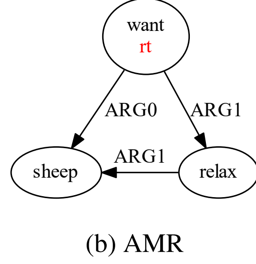


Figure 4: Lexicon. ** can be replaced by a label of the right category. Examples: G_{tv}: love, G_{iv}: sleep; G_{unacc}: relax; G_{mod}: red; G_{n}: prince,rose,sheep,pilot; G_{scv}: want; G_{ocv}: persuade; $G_{\mathrm{c\text{-}[s]}}$: and (seeking operands of type [S]). rt stands for *root*, S for *subject*, O for *object*, and mod for *modifier*.

Upon closer reflection, the structure of the HR term in Fig. 2b is not arbitrary. Many semantic theories assume that two key operations in combining semantic representations compositionally are *application* (i.e., the combination of a predicate with a complement) and *modification*. The term in Fig. 2b simply spells out application for the O-argument of "love": The root of "rose" is inserted into the O-argument-slot, and afterwards we forget the O-source since the slot has been filled. Here we define the *apply-modify (AM) graph algebra*, which replaces the rename-merge-forget operation sequences of the HR algebra with operations that directly model application and modification. In this way, we constrain the set of possible terms for each graph, while preserving linguistically motivated compositional structures. For instance, there will be no equivalent for the term in Fig. 3b.

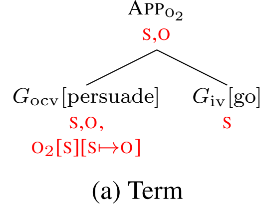


(a) Term

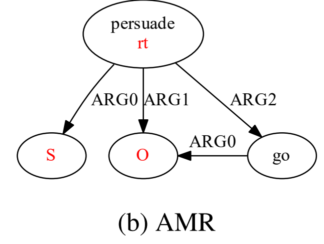


(b) AMR

Figure 5: Subject control *The sheep wants to relax*



(a) Term



(b) AMR

Figure 6: Object control: *persuade _ to go*

3.2.1 Application

We first define the *apply* operation App_α, where α is a source name. In the simple case of Fig. 2a, App_{O} renames the rt-source of its second argument G_2 to O, merges the result with the first argument G_1, and then forgets O – just as in the HR term in Fig. 2b, but with a single algebra operation.

In this simple case, G_2 was complete; its only source was rt. However, in certain cases, we want to combine a predicate with an argument that is itself still looking for arguments. Take the graph in Fig. 5b for example, corresponding to the sentence "the sheep wants to relax", where the sheep is both the wanter and the relaxer. For the subject control verb *want*, we use the lexicon entry $G_{\text{scv}}[\text{want}]$ of Fig. 4. Its O source is *annotated* with the *argument type* S (written O[S]). This means that $G_{\text{scv}}[\text{want}]$ requires its object argument to contain an S-source; during application this node is merged with the S-source of $G_{\text{scv}}[\text{want}]$ itself. This yields a graph with an (undirected) cycle, that is, a graph that is not a tree.

We also allow annotations for *renaming* sources, in order to model phenomena such as object control verbs, as in "the prince persuaded the sheep to sleep" (see Fig. 6a-b). Here, *sheep* is both the subject of *sleep* and the object of *persuade*. We can handle this with the graph for $G_{\text{ocv}}[\text{persuade}]$ in Fig. 4, that features an O_2-source which is annotated as $\text{O}_2[\text{S}][\text{S}{\mapsto}\text{O}]$. This O_2-source must be filled by a graph G_2 that still has an S-source, which is renamed to an O-source during application, and thus merged with the O-source of persuade. This yields the structure shown in Fig. 6b. To capture these intuitions formally, we present the following definitions.

Definition 3.1 (Graph types (TY)). A *graph type* is a pair $\tau = (T, R)$ of a function $T : S \to \text{TY}$, where $S \subseteq \mathcal{S}$ is a set of source names, that assigns a graph type to each source, and a function $R : S \to \{r : \mathcal{S} \rightsquigarrow \mathcal{S} \mid r \text{ partial, injective}\}$ that annotates each source with a renaming function. R may only rename sources T requires , i.e. we demand $\forall T(\alpha) = \langle T', R' \rangle, \mathcal{D}(R(\alpha)) \subseteq \mathcal{D}(T')$. We say that S is the *domain* of τ. Intuitively, the graph type τ provides annotations for all source names in S.

Definition 3.2 (Annotated s-graph (as-graph)). An *annotated s-graph (as-graph)* is a pair $\mathcal{G} = \langle G, \tau \rangle$ of an s-graph $G = (g, S)$ that contains a "root" source (i.e. $\text{rt} \in \mathcal{D}(S)$) and a *graph type* $\tau \in \text{TY}$ with domain $S \setminus \{\text{rt}\}$. We write $\mathcal{AS}$ for the set of all as-graphs.

Our notation, as seen in the above examples, follows the pattern $\alpha[T(\alpha)][R(\alpha)]$ for a source α and its annotation, but we simplify it and drop empty types and functions. For example, the notation O in $G_{\text{tv}}[\text{love}]$ indicates that $T(\text{O}) = (\emptyset, \emptyset)$ and $R(\text{O}) = \emptyset$. The notation O[S] in $G_{\text{scv}}[\text{want}]$ indicates that $T(\text{O}) = (\{\text{S} \mapsto (\emptyset, \emptyset)\}, \{\text{S} \mapsto \emptyset\})$ and $R(\text{O}) = \emptyset$. That is, we require the argument to have an S-source that itself is not further annotated, and we do not rename it. Finally, the notation $\text{O}_2[\text{S}][\text{S}{\mapsto}\text{O}]$ in $G_{\text{ocv}}[\text{persuade}]$ indicates that similarly $T(\text{O}_2) = (\{\text{S} \mapsto (\emptyset, \emptyset)\}, \{\text{S} \mapsto \emptyset\})$, but now $R(\text{O}_2) = \{\text{S} \mapsto \text{O}\}$, signalling the rename.

Definition 3.3 (Apply operation (App)). Let $\mathcal{G}_1 = ((g_1, S_1), (T_1, R_1))$, $\mathcal{G}_2 = ((g_2, S_2), (T_2, R_2))$ be as-graphs. Then we let $\text{App}_\alpha(\mathcal{G}_1, \mathcal{G}_2) = ((g', S'), (T', R'))$ such that

$$(g', S') = f_\alpha((g_1, S_1) \mid\mid \text{ren}_{\{\text{rt} \mapsto \alpha\}}(\text{ren}_{R_1(\alpha)}((g_2, S_2))))$$
$$T' = (T_1 \setminus \{\alpha\}) \cup (T_2 \circ \overline{R_1(\alpha)^{-1}})$$
$$R' = (R_1 \setminus \{\alpha\}) \cup (R_2 \circ \overline{R_1(\alpha)^{-1}})$$

if and only if

1. $\mathcal{G}_1$ actually has an α-source to fill, i.e. $\alpha \in \mathcal{D}(T_1)$,
2. $\mathcal{G}_2$ has the type α is looking for, i.e. $T_1(\alpha) = (T_2, R_2)$, and
3. T', R' are well-defined (partial) functions;

otherwise $\text{APP}_\alpha(\mathcal{G}_1, \mathcal{G}_2)$ is undefined.

The interpretation is just as discussed at the start of Section 3.2 above: we apply all renamings required by R_1 to (g_2, S_2), we rename the root to α, we merge the graphs, and then we forget α. The type of the output graph, (T', R'), is defined such that the source we just filled, α, is removed, and the renaming function of $\mathcal{G}_1$ at α is applied to the domains of T_2 and R_2, so that any requirements $\mathcal{G}_2$ had on its arguments are properly carried over into the new renamed graph. Conditions 1 and 2 ensure that the operation matches the intuition behind the source annotations. Condition 3 guarantees that there are no conflicts in the remaining source annotations of the two graphs. Note that since $T_1(\alpha)$ equals the type (T_2, R_2) if Condition 2 holds, the type (T_1, R_1) can then alone guarantee Condition 3. Observe that the term in Fig. 2a generates the as-graph in Fig. 2f; in both Figures 5 and 6, the term in (a) generates the graph in (b).

3.2.2 Modification

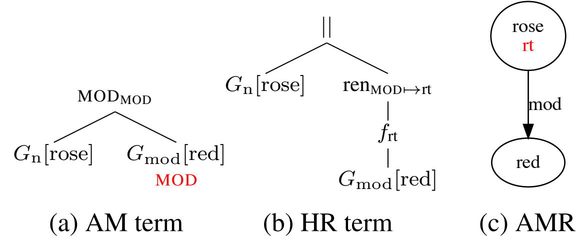


Figure 7: Modification: *a red rose*

We further define a *modify* operation MOD_α, which models modification of its first argument G_1 by its second argument G_2. An example of using MOD_{MOD} to construct an as-graph for "red rose" is shown in Fig. 7, where the modify operation captures the HR term in Fig. 7b: We forget the rt-source of the as-graph $G_{\text{mod}}[\text{red}]$; rename its MOD-source to rt; and then merge it with $G_{\text{n}}[\text{rose}]$. That is, we shift the rt-source of the modifer G_2 to the unlabelled MOD-source and attach it at the root of G_1. This yields the AMR in Fig. 7c. Unlike in the apply case, we can repeat this modification operation as many times as we like: no sources of G_1 are forgotten.

Definition 3.4 (Modify operation (MOD)). In general, we define the *modify operation* for a source α as follows. Again, let $\mathcal{G}_1 = ((g_1, S_1), (T_1, R_1))$, $\mathcal{G}_2 = ((g_2, S_2), (T_2, R_2))$ be as-graphs. Then we let $\text{MOD}_\alpha(\mathcal{G}_1, \mathcal{G}_2) = ((g', S'), (T_1, R_1))$ such that

$$(g', S') = (g_1, S_1) \mid\mid \text{ren}_{\{\alpha \mapsto \text{rt}\}}(f_{\text{rt}}((g_2, S_2)))$$

if and only if

1. $\alpha \in \mathcal{D}(\tau_2)$, i.e. $\mathcal{G}_2$ has an α source,
2. $T_2(\alpha) = (\emptyset, \emptyset)$, i.e. $\mathcal{G}_2$ does not have complex expectations at α, and
3. $T_2 \setminus \alpha \subseteq T_1$ and $R_2 \setminus \alpha \subseteq R_1$, i.e. any remaining sources and annotations in $\mathcal{G}_2$ are already in $\mathcal{G}_1$;

otherwise it is undefined.

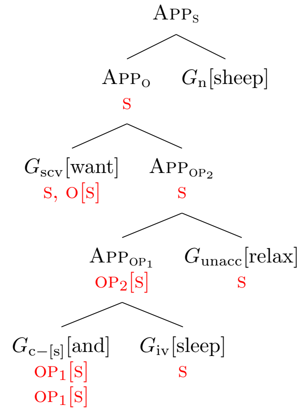
Again, the s-graph evaluation and Condition 1 are straightforward. Modification is more restricted then application, and we demand that the modifier does not change the modifiee's type (Condition 3). We do however allow additional sources in $\mathcal{G}_2$ to merge with existing ones of $\mathcal{G}_1$. For example, when *chew* would modify *swallow* to create the graph in Fig. 1 ("without chewing"), their subject and object would merge. Condition 2 avoids using e.g. the control structure of $G_{\text{scv}}[\text{want}]$ for modification.

We conclude this section by defining the *apply-modify graph algebra (AM algebra)* as an algebra whose domain is the set of all as-graphs. In addition to constants (which evaluate to as-graphs), the AM algebra's signature contains the symbols APP_α (of rank 2) and MOD_α (of rank 2). The associated functions are the ones just defined.

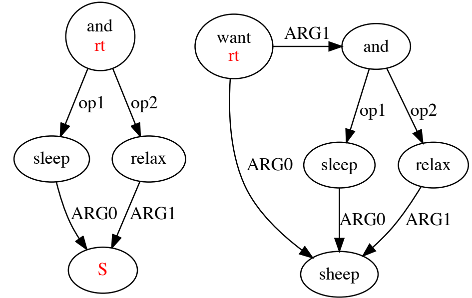
4 Linguistic Discussion

The AM algebra restricts the derivations for a given AMR. The danger, then, is that we could lose all derivations for an AMR, making it unparseable, or that the terms we are left with are not linguistically reasonable. In this section, we show we find reasonable terms for a range of challenging examples. A quantitative analysis of the amount of graphs in the AMRBank for which we can find a decomposition is provided in Section 6.

We have already seen how to derive simple argument application, modification, and control constructions. APP is designed explicitly to parallel for example beta-reduction in lambda calculus, and syntactic operations such as forward and backward application in categorial grammars or endocentric context-free rules. The two arguments of the function combine in such a way that one is in a sense inserted into the other, and the operation is only permitted if the types are correct. In an AM-algebra, APP_α is only allowed if the first argument's type includes the source α, and the second argument's type is $T_1(\alpha)$, and the result is a graph in which the first argument keeps its original root, and the second graph is inside the first. MOD is designed to parallel for example modification of phrases by phrases in a context-free grammar, or modification as X/X categories in categorial grammars: the type of the modifier is a subset of the modified graph, so that modification has no effect on the type of the modified graph, and the modification happens at the root. Modification can also derive control in *secondary predicates* that modify the verb phrase and link an argument to an argument of the verb. For example, to derive (1), the graph for *without dreaming* modifies $G_{\text{iv}}[\text{sleep}]$ while both have an open S-source.



(a) Term



(b) *sleep and relax* (c) *The sheep wants to sleep and relax.*

Figure 8: Conjoining intransitive verbs

(1) The prince$_i$ slept [without [$___i$ dreaming]]

4.1 Coordination

Coordination is a source of re-entrancies in AMRs. For example, when two verb phrases are conjoined, as in (2-a), their subjects must co-refer. Objects can also co-refer in English, as in (2-b). Control verbs, which already have re-entrencies of their own, can be conjoined, as in (2-c). Even subject- and object-control verbs can be conjoined if the object control verb is in the passive (2-d).

(2) a. The prince$_i$ $___i$ sang and $___i$ danced
b. The prince$_i$ $___i$ grew $___j$ and $___i$ loved $___j$ a rose$_j$
c. The sheep$_i$ $___i$ wanted and $___i$ needed $___i$ to relax
d. The prince$_j$ wanted $___j$ to go$_v$, or $___j$ was persuaded $___j$ to $___v$.
e. The rose$_i$ [asked $___j$ $___v$] and $___i$ [persuaded the Prince$_j$ to stay$_v$].

Coordination is generally observed to be between like things; for us this mean the arguments have the same type. For example, in Fig. 8, we choose an *and* that chooses arguments that are missing their subject – it has annotated sources $\text{OP}_i[\text{S}]$. When $G_{\text{iv}}[\text{sleep}]$ and $G_{\text{unacc}}[\text{relax}]$ merge, so do their subjects. In this way, the graph for *sleep and relax* can be selected by a control verb, $G_{\text{scv}}[\text{want}]$, merging its subject with theirs. Similarly, for example (2-e), *ask* and *persuade* are conjoined by a conjunction *and* which is looking for two object-control verbs; that is, *and* has type $\{\text{OP}_1[\text{S,O,O}_2[\text{S}][\text{S}{\mapsto}\text{O}]], \text{OP}_2[\text{S,O,O}_2[\text{S}][\text{S}{\mapsto}\text{O}]]\}$.

There is nothing in the algebra that principally prevents coordination of graphs with different types; however, we restrict our lexicon to constants for coordination nodes that expect like types in their

arguments – we do this in our implementation in Section 5.

4.2 Relative Clauses

Relative clauses are unusual in that one of the arguments of a modifier is the very thing it is modifying. For example, in (3-a), the relative clause *that relaxed* has *sheep* as the subject of *relax*, and *that relaxed* modifies *sheep*. To capture this, we include MOD_S and MOD_O in our repertoire. For a subject relative we make the subject into the root and use it to modify *sheep*, as in Figure 9.

(3) a. [The sheep$_i$ [that $___i$ relaxed]] $___i$ slept
 b. [The asteroid$_i$ that the pilot thought the prince visited $___i$] is tiny

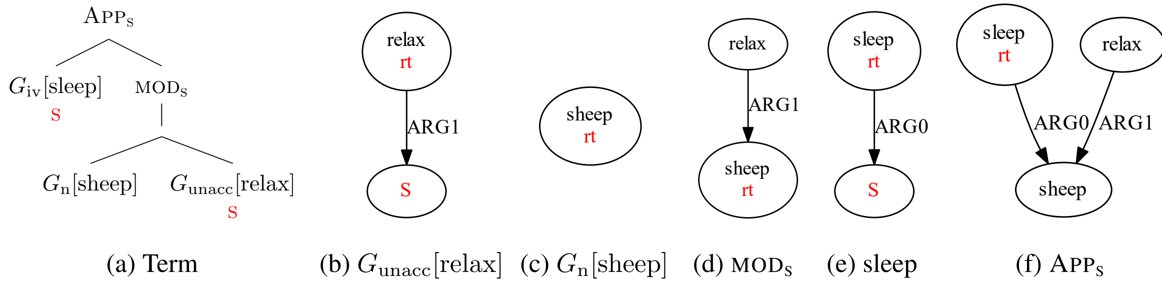


Figure 9: Relative Clause: *The sheep that relaxed slept*

An unboundedly embedded argument can be relativised on, as in (3-b). We handle these the same way they are handled in Tree Adjoining Grammars: by relativising on the clausal argument slot (Fig.10).

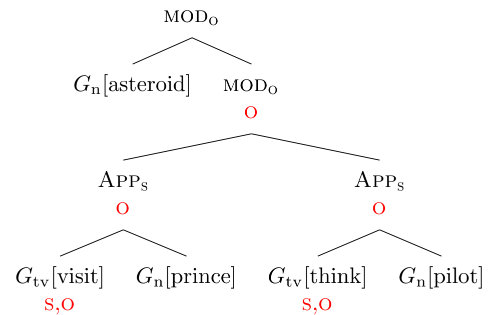


Figure 10: *the asteroid the pilot thinks the prince visited __*

5 Decomposing AMRs with the AM algebra

At this point, we have defined the AM algebra – as a more constrained algebra of graphs than the HR algebra – and shown the adequacy of the apply and modify operations for a number of nontrivial linguistic examples. We will now show how to enumerate AM terms that evaluate to a given graph, e.g. an AMR in the AMRBank. As indicated above, this is a crucial ingredient for grammar induction.

The first step in decomposing a graph G in this way is to select the constants for as-graphs that we will use in the AM algebra – i.e., "atomic" as-graphs such as those in Fig. 4. During grammar induction, we have no grammar or lexicon to draw from, so we will use heuristic methods to extract constants from G. Throughout, we assume that G is an AMR, and we will use a fixed set of sources $\mathcal{S} = \{\text{rt}, \text{S}, \text{O}, \text{O}_2, \ldots, \text{O}_9, \text{MOD}, \text{POSS}, \text{DOMAIN}\} \cup \{\text{OP}_x \mid \text{op}_x \text{ edge label occurs in the corpus}\}$.

5.1 Constants and their types

We start by cutting G up into the subgraphs that will serve as graph backbones of the constants. We do this by splitting G into *blobs*. A blob consists of a main labeled node and its *blob edges*, which are the node's outgoing edges with an ARG_x, op_x, snt_x ($x \in \mathbb{N}$), domain, poss or part label, and its incoming edges with any other label. Blobs

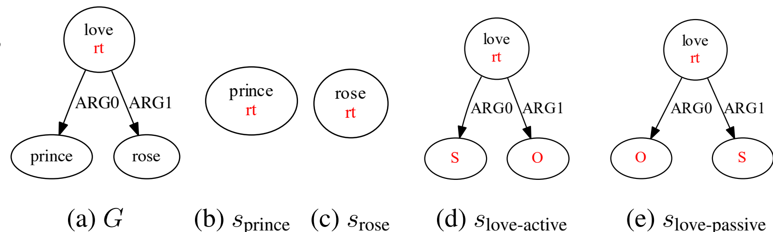


Figure 11: (a) An AMR G; (b)-(e) the constants we obtain

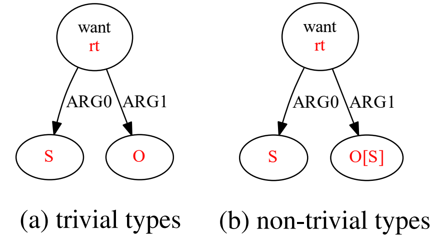
defined in this way uniquely partition an AMR's edge set. An example of an AMR's blobs is shown in Fig. 1, where the blobs are distinguished by colour. For example, the *chew* blob is the red subgraph, including unlabelled nodes where ever a red edge touches a non-red node. These unlabelled endpoints are its *blob-targets*. We will construct a set of constants for each blob, such that the value of each constant is an as-graph whose graph component is the blob. The main node of the blob will be the rt-source. It remains to assign source names to the blob-targets and annotate them with types and renaming functions. The different choices of annotated source names constitute the different constants for this blob.

5.1.1 Source names

We heuristically assign (syntactic) source names from $\mathcal{S}$ to the blob-target nodes based on the edge label of their adjacent edge in the blob. Let v be a node. Canonically, we use the following edge-to-source mapping E2S to determine sources for v's blob-targets: For most nodes v, E2S maps ARG0 to S; ARG1 to O and other ARG$_x$ to O$_x$; poss and part to POSS; snt$_x$, op$_x$ and domain to themselves; and all other edges to MOD. Exceptionally, if v has a node label that is a conjunction[2] and at least two outgoing ARG$_x$ edges, we map ARG$_x$ to OP$_x$ instead. E2S determines the canonical *target-to-source mapping* b_v, which assigns a source to each blob-target u: if the edge between v and u has label e, $b_v(u) = \text{E2S}(e)$. When decomposing the graph in Figure 11a, looking at the *love* node as v, this gives us the constant in Figure 11d.

A given blob may generate more than one constant, each with different sources on different nodes; accordingly, for each node v in G, we collect a *set* $B(v)$ of such target-to-source mappings. $B(v)$ contains the canonical mapping b_v, and we generate further target-to-source mappings by applying a fixed set of lexical rules to b_v. The *passive* rule switches S with any O, and *object promotion* maps O$_i$ to O$_{i-1}$ (let O$_0$=O). We allow all results of such mappings with at most one use of *passive* that have no duplicate source names. For example, the constant in Figure 11e is a result of the passive rule. For each mapping in $B(v)$, we create a constant with the respective sources and trivial types.

5.1.2 Annotations



(a) trivial types (b) non-trivial types

Figure 12: Possible source assignments for the *want* constant for the graph in Figure 5b.

We can also use these target-to-source mappings to extract constants that have sources with non-trivial argument types and renaming functions. Consider the subject-control AMR in Fig. 5b in section 3.2.1 above. So far, we obtain the constant in Fig. 12a, but we also want to generate the constant $G_{\text{scv}}[\text{want}]$ in Fig. 12b; i.e. determine the S entry in $T(\text{O})$. Writing v_{want}, v_{sheep}, and v_{relax} for the *want*, *sheep*, and *relax* nodes of the graph in 5b, note that it is the ARG1 edge from v_{relax} to v_{sheep} that signals the control structure. That is, v_{want} has a blob-target v_{relax}, and the two share a *common* blob-target v_{sheep}. For such a triangle structure, we consider any target-to-source mappings $m_w \in B(v_{\text{want}})$ and $m_r \in B(v_{\text{relax}})$. We then add a constant for v_{want} which as before uses the source names of m_w, but now the annotation of $m_w(v_{\text{relax}})$ has an entry for $m_r(v_{\text{sheep}})$, anticipating the open source coming from the v_{relax} constant. We add a rename annotation $[m_r(v_{\text{sheep}}) \mapsto m_w(v_{\text{sheep}})]$ if necessary. That is, we set up the annotation in the v_{want} constant such that when we apply it to a v_{relax} constant that has sources according to m_r, we obtain the structure we found in the graph. Take for example $m_w = \{v_{\text{relax}} \mapsto \text{O}, v_{\text{sheep}} \mapsto \text{S}\}$ and $m_r = \{v_{\text{sheep}} \mapsto \text{S}\}$.[3] In this case, $m_w(v_{\text{sheep}}) = m_r(v_{\text{sheep}}) = \text{S}$, therefore no rename is necessary and we obtain the constant of Figure 12b. If we choose $m_r = \{v_{\text{sheep}} \mapsto \text{O}\}$ instead, we obtain a constant for the v_{want} blob where the O source is annotated O[O][O$\mapsto$S]. In this graph, this is not particularly meaningful from a linguistic perspective, but in other graphs this principle allows us to generate e.g. the object control structure of

[2] According to the AMR documentation, these are *and*, *or*, *contrast-01*, *either* and *neither*.

[3] Since *relax* is an unaccusative verb, its sole argument is semantically an object (ARG1) but we can treat it as a syntactic subject by choosing the passive mapping, which promotes the object to subject.

$G_{\text{ocv}}[\text{persuade}]$. To ensure that we recover the correct constant, we simply add constants for all choices of $m_w \in B(v_{\mathsf{want}})$ and $m_r \in B(v_{\mathsf{relax}})$.

Let us now find the constants for the *and* node in Fig. 8b. Our algorithm restricts constants to coordination of like types. In the intended AM term, shown in Fig. 8a, we first coordinate *relax* and *sleep* before we apply the result to the common argument *sheep*. To generate the constant for *and*, we consider maps $m_s \in B(v_{\mathsf{sleep}})$ and $m_r \in B(v_{\mathsf{relax}})$, where v_{sleep} and v_{relax} are the nodes labelled *sleep* and *relax* respectively. The *sheep* node v_{sheep} is a blob-target of both v_{sleep} and v_{relax}. If additionally the target-to-source maps agree, e.g. $m_s(v_{\mathsf{sheep}}) = m_r(v_{\mathsf{sheep}}) = \textsc{s}$, we add a new constant for the *and* blob where both $T(\textsc{op}_1)$ and $T(\textsc{op}_2)$ have an $\textsc{s}$ entry. This yields $G_{\text{c}-[\text{s}]}[\text{and}]$ as depicted in Fig. 4. For the case where $m_s(v_{\mathsf{sheep}}) = \textsc{s}$ but $m_r(v_{\mathsf{sheep}}) = \textsc{o}$, we do not create a new constant. Again, we take all combinations of choices for m_s and m_r into account. We never rename for coordination.

Similar patterns allow us to find possible raised subjects for raising constructions, and to handle coordination of control verbs. Using these patterns recursively, we can handle nested control, coordination and raising constructions. For example in Fig. 8c, finding the *sheep* node as a common target in coordination allows us to generate $G_{\text{scv}}[\text{want}]$ analogously to Fig. 5b.

In sum, we obtain types and renaming functions that cover a variety of phenomena, in particular the ones described in Section 4.

5.2 Coreference

In the AMRBank annotations, the same node can become the argument of multiple predicates in two very different ways: because the grammar specifies it (as with control, (4-b)), and through accidental coreference (4-a).

(4) a. Mary$_i$ thinks she$_{i/j}$'s a genius
  b. Mary$_i$ wants $___i$ to be a genius

Because accidental coreference is not a compositional phenomenon, we add an extra mechanism for handling it. We follow Koller (2015) in introducing special sources $\textsc{coref}1$, $\textsc{coref}2, \ldots, \textsc{coref}n$ for some maximal $n \in \mathbb{N}$. We add variants of the previously found constants with a COREF source at their root. We further add constants consisting of a single unlabelled node, which is both a rt-source and a COREF-source. The COREF sources are never annotated and are ignored in the types. They are never forgotten, and each can therefore be used only on one node in the derivation. Two COREF sources with the same index will be automatically merged together during the usual APP and MOD operations, due to the semantics of the underlying merge operation of the HR algebra.

COREF sources increase runtimes and the number of possible terms per graph significantly (see Section 6), and thus we limit the number of COREF sources to zero to two in practice.

5.3 Obtaining the set of terms

We can compactly represent the set of all AM terms that evaluate to a given AMR G in a *decomposition automaton* (Koller and Kuhlmann, 2011), a chart-like data structure in which shared subterms are represented only once. We can enumerate the terms from this automaton.

To enumerate all rules of the decomposition automaton, we explore it bottom-up, with Algorithm 1. We first find all constants for G in Line 2, as described in Section 5.1, and then repeatedly apply APP and MOD operations (Lines 3 onward; the set $\mathcal{O}$ contains all relevant APP and MOD operations). The constants and the successful operation applications are stored as rules in the automaton.

To ensure that the resulting terms evaluate to the input graph G, we use subgraphs of G as states – like one uses spans in string parsing. This is paired with additional constraints, for example in $\textsc{App}_\alpha(s, s')$, the root of s' must be the same node of G as the α-source node in s. These additional constraints are as described in Groschwitz et al. (2015), when interpreting the AM operations as terms of the HR algebra (c.f. Section 3.2); plus the constraint that a rt-source at the root node of G may not be renamed or forgotten. These constraints are the analogue of only combining neighbouring spans in string parsing.

Let us decompose the graph G in Figure 11a as an example. Let us call the nodes labelled "love", "prince", and "rose" v_{love}, v_{prince} and v_{rose} respectively. In Line 2, we add the subgraphs of Fig. 11(b-e) to the agenda. Say we first pull s_{rose} from the agenda – since the chart is empty at this point, no operation is applicable. Say we pull $s_{\text{love-active}}$ next, and try to combine it with the items in the chart – just s_{rose} at this point. If we try to apply APP_{S}, we realize that this tries to fill the node v_{prince} of $s_{\text{love-active}}$, but the root of s_{rose} is v_{rose}. Thus, the operation fails. (Trying APP_{S} with s_{rose} as the left and $s_{\text{love-active}}$ as the right child fails immediately since s_{rose} has no S-source). MOD_{S} fails similarly. However, APP_{O} succeeds – both the O-source in $s_{\text{love-active}}$ and the rt-source in s_{rose} are at v_{rose} – and produces the graph in Fig. 2f. MOD_{O} fails, since it would involve forgetting rt at v_{love}, and the root of the full graph must be preserved. We therefore add the result of APP_{O} to the agenda and move on.

Algorithm 1 Agenda-chart-algorithm

1: init chart, agenda empty
2: add constants to agenda
3: **while** agenda not empty **do**
4: pull subgraph s from agenda
5: **for** operation $o \in \mathcal{O}$ **do**
6: **for** subgraph s' in chart **do**
7: **if** $o(s, s')$ allowed **then**
8: add $o(s, s')$ to agenda
9: **end if**
10: **if** $o(s', s)$ allowed **then**
11: add $o(s', s)$ to agenda
12: **end if**
13: **end for**
14: **end for**
15: add s to chart
16: **end while**

To explore the possibilities for combining as-graphs efficiently, we do not iterate over all graphs in Line 6, but for each operation use an indexing structure based on source nodes and types.

Note that the operations in the automaton are restricted by the AM algebra's type system. Therefore, selecting the correct constants as in Section 5.1 is critical to obtaining the desired derivations. We do obtain all the terms in the examples in this paper in practice.

6 Evaluation

We conclude by analyzing whether the AM algebra achieves our goal of reducing the number of possible terms for a given AMR, compared to the HR algebra. Both algorithms are implemented and available in the Alto framework[4]. For the HR algebra, we use the setup of Groschwitz et al. (2015): Constants consist of single labeled nodes and single edges, and they are combined using the operations of the HR algebra. We use an HR algebra with two source names (HR-S2) and one with three source names (HR-S3); this has an impact on the set of graphs that can be analyzed and on the runtime complexity. For the AM algebra, we use the method of Section 5 with different numbers of allowed COREF sources (AM-C0, AM-C1, AM-C2 for 0, 1, 2 COREF sources respectively). We use all graphs of the LDC2016E25 training corpus with up to 50 nodes, for a total of 35685 graphs.

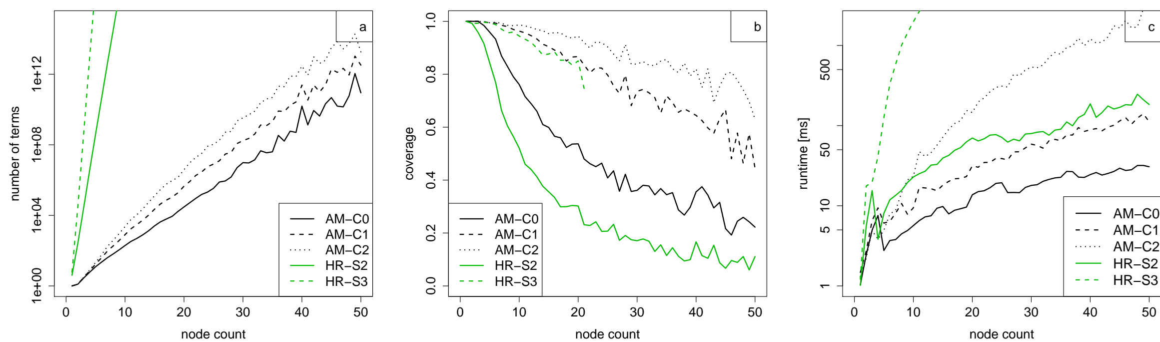


Figure 13: Number of terms per AMR (a), coverage (b) and runtimes (c).

Coverage. Consider first the *coverage* of the different graph algebras, i.e. the proportion of graphs of a given size for which we find at least one term, shown in Fig. 13b as a function of the graph size. As expected, coverage goes up as the number of source nodes (for HR) and COREF nodes (for AM)

[4]bitbucket.org/tclup/alto

increases. The coverage of AM-C0 is higher than that of HR-S2 because HR-S2 can only analyze graphs of treewidth 1, i.e. without (undirected) cycles, whereas AM-C0 can handle local re-entrancies e.g. from control constructions through the type annotations. For example, the AMRs in Fig. 5b,6b can be decomposed by AM-C0 and HR-S3, but not HR-S2. The highest coverage is achieved by AM-C2.

Number of terms. We now turn to the (geometric) mean number of terms each algebra assigns to those graphs of a given size that it can analyze (Fig. 13a). We find that the AM algebras achieve a dramatic reduction in the number of terms, compared to the HR algebras: Even the high-coverage AM-C2 has much fewer terms than the very low-coverage HR-S2 (note the log-scale on the vertical axis). As an example, switching from HR-S2 to AM-C0 reduces the number of terms for the graph in Fig. 3a from 3584 to 4 (they differ in active vs passive, and order of application). For 5 nodes, the average for HR-S3 is 10^{17} terms, and for AM-C2 just 21. This reduction has multiple reasons: we can use larger constants in the AM algebra, and the graph-combining operations of the AM algebra are much more constrained. Further, the type system and carefully chosen set of constants restrict application and modification.

Note that just because an algebra can find *some* term for an AMR does not necessarily mean that it makes sense from a linguistic perspective (cf. Fig. 3b). Conversely, by reducing the set of possible terms, there is a risk that we might throw out the linguistically correct analysis. By choosing the operations of the AM algebra to match linguistic intuitions about predicate-argument structure, we have reduced this risk. We leave a precise quantitative analysis, e.g. in the context of grammar induction, for future work.

Runtime. We finish by measuring the mean runtimes to compute the decomposition automata (Fig. 13c). Once again, we find that the AM algebra solidly outperforms the HR algebra. The runtimes of HR-S3 are too slow to be useful in practice, whereas even the highest-coverage algebra AM-C2 decomposes even large graphs in seconds. Moreover, the runtimes for AM-C1 are faster than even for the very low-coverage HR-S2 algebra.

The previously fastest parser for graphs using hyperedge replacement grammars was the one of Groschwitz et al. (2016), which used Interpreted Regular Tree Grammars (IRTGs) (Koller and Kuhlmann, 2011) together with the HR algebra. Because we have seen how to compute decomposition automata for the AM algebra in Section 5, we can do graph parsing with IRTGs over the AM algebra instead. The fact that decomposition automata for the AM algebra are smaller and faster to compute promises a further speed-up for graph parsing as well, making wide-coverage graph parsing for large graphs feasible.

7 Conclusion

In this paper, we have introduced the apply-modify (AM) algebra for graphs. The AM algebra replaces the general-purpose, low-level operations of the HR algebra by high-level operations that are specifically designed to combine semantic representations of syntactic heads with arguments and modifiers. We have demonstrated that the AM algebra dramatically reduces the number of terms for given AMR graphs, while supporting natural analyses of a number of challenging linguistic phenomena.

With this work we have laid the foundation for automatically inducing grammars that can map compositionally between strings and AMRs while using linguistically meaningful graph-combining operations. Our immediate next step will be to use the AM algebra for this purpose. On a more theoretical level, while the algebra objects differ greatly, the similarity of the *signature* of the AM algebra with that of the "semantic algebra" of Copestake et al. (2001) is striking. We will explore this connection, and investigate whether a universal signature for a semantic construction algebra can be defined.

8 Acknowledgements

We thank the anonymous reviewers for their comments. We would also like to thank Christoph Teichmann, Antoine Venant and Mark Steedman for helpful discussions. This work was supported by the DFG grant KO 2916/2-1, and a Macquarie University Research Excellence Scholarship for Jonas Groschwitz.

References

Artzi, Y., K. Lee, and L. Zettlemoyer (2015). Broad-coverage ccg semantic parsing with amr. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1699–1710.

Banarescu, L., C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider (2013). Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

Chiang, D., J. Andreas, D. Bauer, K. M. Hermann, B. Jones, and K. Knight (2013). Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*.

Copestake, A., A. Lascarides, and D. Flickinger (2001). An algebra for semantic construction in constraint-based grammars. In *Proceedings of the 39th ACL*.

Courcelle, B. (1993). Graph grammars, monadic second-order logic and the theory of graph minors. In N. Robertson and P. Seymour (Eds.), *Graph Structure Theory*, pp. 565—590. AMS.

Drewes, F., H.-J. Kreowski, and A. Habel (1997). Hyperedge replacement graph grammars. pp. 95–162.

Flanigan, J., S. Thomson, J. Carbonell, C. Dyer, and N. A. Smith (2014). A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1426–1436.

Groschwitz, J., A. Koller, and M. Johnson (2016). Efficient techniques for parsing with tree automata. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

Groschwitz, J., A. Koller, and C. Teichmann (2015). Graph parsing with S-graph Grammars. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*.

Jones, B., J. Andreas, D. Bauer, K.-M. Hermann, and K. Knight (2012). Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of COLING*.

Jones, B. K., S. Goldwater, and M. Johnson (2013). Modeling graph languages with grammars extracted via tree decompositions. In *Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing*, pp. 54–62.

Koller, A. (2015). Semantic construction with graph grammars. In *Proceedings of the 11th International Conference on Computational Semantics*, pp. 228–238.

Koller, A. and M. Kuhlmann (2011). A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies*.

Misra, D. K. and Y. Artzi (2016). Neural shift-reduce ccg semantic parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*.

Peng, X., L. Song, and D. Gildea (2015). A synchronous hyperedge replacement grammar based approach for amr parsing. In *Proceedings of the 19th Conference on Computational Language Learning*, pp. 32–41.

Peng, X., C. Wang, D. Gildea, and N. Xue (2017). Addressing the data sparsity issue in neural AMR parsing. In *Proceedings of the 15th EACL*.

Steedman, M. (2001). *The Syntactic Process*. Cambridge, MA: MIT Press.