Compiling HPSG constraint grammars into logic programs

Thilo Götz

SFB 340, University of Tübingen, Germany*

Abstract. This paper defines a translation from HPSG constraint grammars into constraint logic programs that preserves the parsing problem. We will show that there can be no such complete translation, yet we will argue that for theoretical as well as practical reasons, it is interesting to see how closely one can approximate HPSG grammars with logic programs. We will thus examine the properties of the translation in detail and come up with a restriction on HPSG grammars that ensures the completeness of the translation. An optimised version of the translation procedure has been implemented in Prolog.

1 Introduction

Head-driven Phrase Structure Grammar (HPSG, [15]) has become a de facto standard in certain areas of computational linguistics. Many grammar implementations claim to be HPSG grammars or at least to be based on HPSG. However, there is always a fairly wide gap between the linguistic theory of HPSG, and the implementations of HPSG grammars. This is mainly due to the different logical formalisms employed on the linguistic and the implementational side.

We will base our account of a logical formalism for the linguistic HPSG theory on [11]. In [11], a logic for HPSG is defined as a subset of first-order predicate logic. A HPSG grammar is simply some first-order theory. The *parsing problem*[2] is then defined as a satisfiability problem. Given a theory Θ and some expression ϕ (which codes the string we want to parse), we ask if there is some model of Θ that satisfies ϕ. We call this the *admissibility* problem.

Implementations of HPSG grammars are generally based on logic programming. Now logic programs are also first-order theories, but the parsing problem is different. Given a logic program P and a definite goal G (which again codes the string we want to parse), we ask if P entails G, i.e., if every model of P satisfies G. These two different ways of encoding the parsing problem have recently been discussed in [9].

It would be desirable to have an automatic translation from HPSG grammars to logic programs that preserves the parsing problem for two obvious reasons:

* tg@sfs.nphil.uni-tuebingen.de

[2] The same holds for the generation problem.

it would ensure the faithfulness of the encoding, and it would still enable us to use the wealth of results that have been produced over the last two decades or so for optimising the execution of logic programs. We could also use existing efficiently implemented logic programming systems without having to custom build a new system for the admissibility problem. However, completely abstract considerations tell us that there can be no completely faithful, decidable translation. Since the admissibility problem is undecidable, yet can be reduced to a first-order satisfiability problem, we know that we can enumerate all negative instance, i.e., complete algorithms for solving the *inadmissibility* problem exist [3]. For logic programming, on the other hand, well-known results tell us that only for the positive problem instance do complete algorithms exist (see [13]). We can therefore only give a translation that works in many, but not all cases. In this paper, we consider a candidate translation and show for which class of grammars it is a completely faithful coding. Our approach is a direct encoding of grammar constraints as definite clauses. Finally, we will argue that for most linguistically relevant grammars, the translation does indeed give the desired results.

The rest of the paper is organised as follows. In Sec. 2, we describe the logic that our account is based on. Section 3 then formally defines HPSG grammars and the admissibility problem. In Sec. 4, we define the translation and examine its properties. We close with some concluding remarks in Sec. 5.

2 A Feature Constraint Language

We describe a typed feature logic that can be used as a constraint language in the sense of [8]. The logic is based on those of [12], [16] and [6]. The interested reader is refered to [12] for why this is an appropriate logic for HPSG and a comparison with other approaches.

Definition 1 (signature). A signature is a quadruple $\langle \mathcal{T}, \preceq, \mathcal{F}, \mathrm{approp} \rangle$ s.t.

- $\langle \mathcal{T}, \preceq \rangle$ is a finite join semi-lattice (type hierarchy) with top element $\top$
- $\mathcal{V} = \{t \in \mathcal{T} \mid \text{if } t' \preceq t \text{ then } t' = t\}$
- $\mathcal{F}$ is a finite set of feature names
- $\mathrm{approp} : \mathcal{V} \times \mathcal{F} \rightharpoonup \mathcal{T}$ is a partial function from pairs of minimal types and features to types (appropriateness condition)

The expressions of our language – feature terms and constraints – are built from the symbols in a signature and some decidable set of variables VAR.

Definition 2 (feature terms).

- X is a term if $X \in \mathsf{VAR}$

- t is a term   if $t \in \mathcal{T}$
- $f : T$ is a term   if $f \in \mathcal{F}$ and T is a term.
- $T_1 \wedge T_2$, $T_1 \vee T_2$, $\neg T_1$ are terms   if T_1 and T_2 are terms.[3]

Write Term for the set of terms. In addition to the term language, which is sufficient for expressing HPSG grammars, we need to define *constraints* to be able to embed our logic into the CLP scheme of [8].

Definition 3 (constraints).

- $X = T$ is a constraint if X is a variable and T is a term.
- $F_1 \& F_2$ is a constraint if F_1 and F_2 are constraints.

Definition 4 (interpretation). An interpretation is a triple $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ s.t.

- $\mathcal{U}$ is a set of objects, the domain of $\mathcal{I}$
- $\mathcal{S} : \mathcal{U} \to \mathcal{V}$ is a total function from the set of objects to the set of minimal types
- $\mathcal{A} : \mathcal{F} \to \mathcal{U}^{\mathcal{U}}$ is an attribute interpretation function s.t.
  - for each $u \in \mathcal{U}$, for each $f \in \mathcal{F}$, if $\text{approp}(\mathcal{S}(u), f)$ is defined and $\text{approp}(\mathcal{S}(u), f) = t$, then $(\mathcal{A}(f))(u)$ is defined and $\mathcal{S}((\mathcal{A}(f))(u)) \preceq t$
  - for each $u \in \mathcal{U}$, for each $f \in \mathcal{F}$, if $\mathcal{A}(f)(u)$ is defined, then $\text{approp}(\mathcal{S}(u), f)$ is defined and $\mathcal{S}(\mathcal{A}(f)(u)) \preceq \text{approp}(\mathcal{S}(u), f)$

Features denote partial functions on the domain of interpretation. The domain and range of these functions are determined by the approp function.

Definition 5 (term denotation). Term interpretation is defined with respect to a variable assignment $\alpha \in \mathsf{ASS} = \mathcal{U}^{\mathsf{VAR}}$:

- $[\![X]\!]^{\mathcal{I}}_{\alpha} = \{\alpha(X)\}$ if $X \in \mathsf{VAR}$
- $[\![t]\!]^{\mathcal{I}}_{\alpha} = \{u \in \mathcal{U} \mid \mathcal{S}(u) \preceq t\}$ if $t \in \mathcal{T}$
- $[\![f : T]\!]^{\mathcal{I}}_{\alpha} = \{u \in \mathcal{U} \mid (\mathcal{A}(f))(u) \in [\![T]\!]^{\mathcal{I}}_{\alpha}\}$ if $f \in \mathcal{F}$ and T is a term
- $[\![T_1 \wedge T_2]\!]^{\mathcal{I}}_{\alpha} = [\![T_1]\!]^{\mathcal{I}}_{\alpha} \cap [\![T_2]\!]^{\mathcal{I}}_{\alpha}$ if T_1 and T_2 are terms
- $[\![T_1 \vee T_2]\!]^{\mathcal{I}}_{\alpha} = [\![T_1]\!]^{\mathcal{I}}_{\alpha} \cup [\![T_2]\!]^{\mathcal{I}}_{\alpha}$ if T_1 and T_2 are terms
- $[\![\neg T_1]\!]^{\mathcal{I}}_{\alpha} = \mathcal{U} \backslash [\![T_1]\!]^{\mathcal{I}}_{\alpha}$ if T_1 is a term

Definition 6 (constraint denotation). Constraints are assigned sets of variable assignments as denotations:

[3] Implication is defined as usual: $S \Rightarrow T \equiv \neg S \vee T$.

– $[\![X = T]\!]^{\mathcal{I}} = \{\alpha \in \mathsf{ASS} \mid \alpha(X) \in [\![T]\!]^{\mathcal{I}}_{\alpha}\}$ if $X \in \mathsf{VAR}$ and T is a term
– $[\![T_1 \& T_2]\!]^{\mathcal{I}} = [\![T_1]\!]^{\mathcal{I}} \cap [\![T_2]\!]^{\mathcal{I}}$ if T_1 and T_2 are terms

Our feature logic fulfills all the requirements of [8] for a constraint language. We can therefore simply apply their schema to extend our language with relation symbols and obtain a sound and complete operational semantics for definite clause programs.

Definition 7 ($\mathcal{R}(\mathcal{FL})$).

1. $\mathcal{R}$ is a decidable set of relation symbols
2. the variables of $\mathcal{R}(\mathcal{FL})$ are the variables of $\mathcal{FL}$
3. the constraints of $\mathcal{R}(\mathcal{FL})$ are defined inductively as follows
   (a) every $\mathcal{FL}$ constraint is an $\mathcal{R}(\mathcal{FL})$ constraint
   (b) if $r \in \mathcal{R}$ is a relation symbol with arity n and $\mathbf{x}$ is an n-tuple of pairwise distinct variables, then $r(\mathbf{x})$ is an $\mathcal{R}(\mathcal{FL})$ constraint called an atom
   (c) if F and G are $\mathcal{R}(\mathcal{FL})$ constraints, then so are
      – $\emptyset$ (empty conjunction)
      – F & G (conjunction)
      – $F \rightarrow G$ (implication)
4. a definite clause is an $\mathcal{R}(\mathcal{FL})$ constraint $A \leftarrow \phi \ \& \ B_1 \ \& \ \ldots \ \& \ B_n$, where $A, B_1, \ldots, B_n$ are atoms and ϕ is an $\mathcal{R}(\mathcal{FL})$ constraint
5. an interpretation $\mathcal{A}$ of $\mathcal{R}(\mathcal{FL})$ is obtained from an $\mathcal{FL}$-interpretation $\mathcal{I}$ with domain $\mathcal{U}$ by choosing for every $r \in \mathcal{R}$ a relation $r^{\mathcal{A}}$ on $\mathcal{U}$ with the same arity, and by defining:
   (a) the domain of $\mathcal{A}$ is $\mathcal{U}$
   (b) $[\![\phi]\!]^{\mathcal{A}} = [\![\phi]\!]^{\mathcal{I}}$ if ϕ is an $\mathcal{FL}$ constraint
   (c) $[\![r(\mathbf{x})]\!]^{\mathcal{A}} = \{\alpha \in \mathsf{ASS} \mid \alpha(\mathbf{x}) \in r^{\mathcal{A}}\}$
   (d) $[\![\emptyset]\!]^{\mathcal{A}} = \mathsf{ASS}$, $\quad [\![F \ \& \ G]\!]^{\mathcal{A}} = [\![F]\!]^{\mathcal{A}} \cap [\![G]\!]^{\mathcal{A}}$
   (e) $[\![F \rightarrow G]\!]^{\mathcal{A}} = (\mathsf{ASS} - [\![F]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}}$

Since this is a straightforward application of [8]'s CLP scheme, we refer the reader to that work for the definition of definite goals and solutions, and for the soundness and completeness results.

3 HPSG grammars

We consider an HPSG grammar to be a set of feature terms. The elements of a grammar are considered to be universally quantified in a certain sense: in a model of a grammar, every grammar constraint must be true of every object.

Definition 8 (model). A model of a set $\mathcal{G}$ of feature terms[4] with respect to a signature Σ is an interpretation $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ s.t. for every $u \in \mathcal{U}$ and for every $T \in \mathcal{G}$ there is an $\alpha \in \mathsf{ASS}$ s.t. $u \in [\![T]\!]^{\mathcal{I}}_{\alpha}$.

Definition 9 (admission). $\mathcal{G}$ admits a term T iff there is a model $\mathcal{I}$ of $\mathcal{G}$ and a variable assignment α s.t. $[\![T]\!]^{\mathcal{I}}_{\alpha} \neq \emptyset$.

Admission is a notion of grammaticality for HPSG grammars. We've given a definition of model for arbitrary sets of constraints, but the compiler actually requires a grammar to be in a specific format.

Definition 10 (grammar). A grammar is a pair $\langle \Sigma, R \rangle$ such that Σ is a signature $\langle \mathcal{T}, \preceq, \mathcal{F}, \text{approp} \rangle$ is a signature, $R \subseteq \{t \Rightarrow T \mid t \in \mathcal{T} \text{ and } T \in \mathsf{Term}\}$ and R is finite.

This is not a real restriction, since every term T can be trivially brought into the required format by substituting the equivalent $\top \Rightarrow T$. Of course, it is possible to do this more intelligently in many cases, specifically if the term to be transformed already is an implication. That we can do this is due to the special properties of our feature logic, our classical use of negation and the appropriateness conditions. This doesn't work in all variants of feature logic.

Example 1 (HPSG Head-Feature Principle). [5]

$$\begin{aligned}
&\text{dtrs}: \textit{headed-struc} \;\Rightarrow \\
&\qquad \text{synsem}: \text{loc}: \text{cat}: \text{head}: X \;\wedge \\
&\qquad \text{dtrs}: \text{hd-dtr}: \text{synsem}: \text{loc}: \text{cat}: \text{head}: X
\end{aligned}$$

is equivalent to

$$\begin{aligned}
&\textit{phrase} \;\Rightarrow \\
&\qquad \text{dtrs}: \textit{coord-struc} \;\vee \\
&\qquad \text{synsem}: \text{loc}: \text{cat}: \text{head}: X \;\wedge \\
&\qquad \text{dtrs}: \text{hd-dtr}: \text{synsem}: \text{loc}: \text{cat}: \text{head}: X
\end{aligned}$$

We now define an inheritance function IC that will later simplify our translation.

Definition 11 (IC). $\text{IC} : \mathcal{T} \rightarrow \mathsf{Term}$ is a total function form the set of types to the set of terms s.t. $\text{IC}(t) = \bigwedge_{t \preceq t', t' \Rightarrow T \in R} T$, where the empty conjunction is $\top$. Let $\mathcal{G} = \langle \Sigma, R \rangle$ be a grammar s.t. the sets of free variables of the terms in R are pairwise disjoint. Then $\mathcal{G}_{\text{IC}} = \langle \Sigma, \{t \Rightarrow \text{IC}(t) \mid t \in \mathcal{V} \wedge \text{IC}(t) \neq \top\} \rangle$

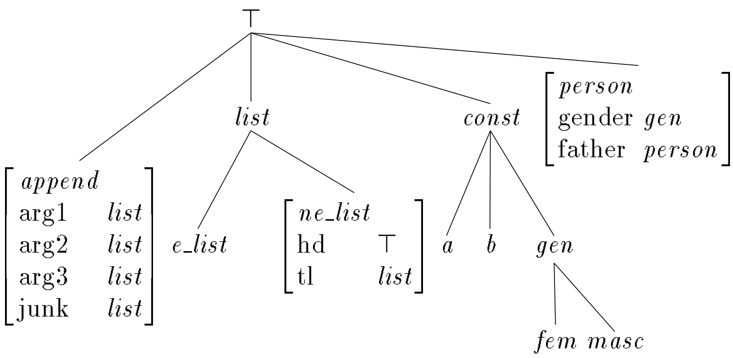
---

[4] Notice that we use the terms of our language to encode grammars, whereas we use the constraints only for the logic programs.

[5] We use the signature from the appendix of [15] with the following notational conventions here: feature names are written in lower case roman font, type names in lower case italics and variables in upper case italics.

Clearly, $\mathcal{G}_{\mathrm{IC}}$ is a grammar equivalent to $\mathcal{G}$.

Example 2 (a simple grammar). This simple grammar shows a junk slot encoding of the append relation.[6] The type hierarchy with the appropriateness conditions is depicted as a graph in the usual manner.



$$
\begin{aligned}
\mathit{append} \Rightarrow (\ & \text{arg1: } \mathit{e\_list} \wedge \text{arg2: } L \wedge \text{arg3: } L \wedge \text{junk: } \mathit{e\_list} \\
\vee\ & \text{arg1: (hd: } H \wedge \text{tl: } T_1) \wedge \text{arg2: } L \wedge \text{arg3: (hd: } H \wedge \text{tl: } T_2) \wedge \\
& \text{junk: (hd: } (\mathit{append} \wedge \text{arg1: } T_1 \wedge \text{arg2: } L \wedge \text{arg3: } T_2) \wedge \text{tl: } \mathit{e\_list})
\end{aligned}
$$

We will use this grammar in the next section to explain the compilation process.

4 From grammars to logic programs

We will now show how a HPSG grammar $\mathcal{G}$ can be translated into a logic program $\mathcal{P}$ s.t. the answers to $\mathcal{P}$ are the terms that $\mathcal{G}$ admits. We will do this by defining a unary relation *gram* that denotes only objects that are licensed by $\mathcal{G}$. Note that we will use some terminology from [8]. Space reasons do not allow us to quote all the relevant passages of that work.

Definition 12 (correct translation). Let $\mathcal{P}$ be a logic program that defines the relation *gram* and $\mathcal{G}$ a grammar. $\mathcal{P}$ is a correct translation of $\mathcal{G}$ iff

$\mathcal{G}$ admits term T iff the goal $\mathit{gram}(X)$ & $X = T$ has a $\mathcal{P}$-answer.

We now partition the set of types $\mathcal{T}$ into three subsets: constrained, hiding and simple types. The motivation for the distinctions is computational and in some sense already an optimisation. In short, it obviates the need for some rather complex partial evaluation later on and reduces the number of choice points.

[6] See [1].

Here's the intuitive meaning of the three sets:

- **constrained types** are exactly those types that unify with some antecedent of the grammar, and whose structure is thus constrained by the conditions stated in the grammar.
- **hiding types** are types that are not constrained themselves but "hide" constrained types somewhere in their structure (loosely speaking).
- **simple types** are neither constrained themselves nor can they hide constrained types anywhere. In other words, the grammar doesn't say anything about them.

Definition 13 (type interaction). We say that two types t and t' interact if they have a common subtype, i.e., $\exists t'' \in \mathcal{T}.\ t'' \preceq t \wedge t'' \preceq t'$.

Definition 14 (directly constrained type). A type is directly constrained just in case it serves as antecedent of an implicational constraint in the grammar, i.e., the set of directly constrained types is $\{t \mid t \Rightarrow T \in R\}$.

The only directly constrained type in Ex. 2 is *append*.

Definition 15 (constrained type). A constrained type is a type that interacts with a directly constrained type. Write C for the set of constrained types.

For Ex. 2, the set of constrained types is then $\{append, \top\}$.

Definition 16 (hiding type). The set of hiding types is the smallest set $H \subseteq \mathcal{T}$ s.t.

1. if t is not a constrained type and $t' \preceq t$, where t' is not a constrained type, $\text{approp}(t', f)$ is defined and $\text{approp}(t', f)$ is a constrained type, then $t \in H$
2. if t is not a constrained type and $t' \preceq t$, where t' is not a constrained type, $\text{approp}(t', f)$ is defined and $\text{approp}(t', f) \in H$, then $t \in H$

4.16 is a recursive definition, the base case of which is a type that has a constrained type appropriate for at least one of its features. The hiding types of Ex. 2 are *list* and *ne_list*.

Definition 17 (simple type). A simple type is a type that is neither a constrained nor a hiding type.

Definition 18 (hiding feature). If t is a constrained or hiding type, then f is a hiding feature on t iff $\text{approp}(t, f)$ is defined and $\text{approp}(t, f)$ is a constrained or hiding type.

The hiding features are the computationally interesting features on a type, as we will see later on. We will now define three different sets of definite clauses defining the *gram* relation, one for simple types, one for hiding types and one for constrained types. Our translation is very direct and rather simple. As we will see in Sec. 4.2, a more complete translation would have to be more complicated and much less efficient.

Definition 19 **($\mathcal{P}_s(\mathcal{G})$).** Let $\mathcal{G}$ be a grammar and S the set of simple types in $\mathcal{G}_{\mathrm{IC}}$. Define

$$\mathcal{P}_s(\mathcal{G}) = \{\mathit{gram}(X) \leftarrow X = t \mid t \in S \wedge t \in \mathcal{V}\}$$

All the clauses for simple types represent base cases of the *gram* relation. From our definition of simple types, it is intuitively clear that any term of a simple type is admissible. An example of a clause generated for a simple type is

$$\mathit{gram}(X) \leftarrow X = \mathit{person}.$$

So despite the cyclic appropriateness conditions of the person type, we are assured that any object of type person is in the gram relation.

Definition 20 **($\mathcal{P}_h(\mathcal{G})$).** Let $\mathcal{G}$ be a grammar and H the set of hiding types in $\mathcal{G}_{\mathrm{IC}}$. For each type $t \in H$, define

$$\begin{aligned} \mathit{Clause}(t) := \mathit{gram}(X) \leftarrow X = (t \wedge f_1\colon Y_1 \wedge \ldots \wedge f_n\colon Y_n)\ \&\\ \mathit{gram}(Y_1)\ \& \ldots \&\ \mathit{gram}(Y_n), \end{aligned}$$

where $f_1 \ldots f_n$ are the hiding features of t and $X, Y_1, \ldots, Y_n$ are pairwise distinct variables. Then let

$$\mathcal{P}_h(\mathcal{G}) = \{\mathit{Clause}(t) \mid t \in H \wedge t \in \mathcal{V}\}$$

This definition introduces a program clause for each minimal hiding type by saying that an object of a hiding type is grammatical just in case the objects under the type's hiding features are also grammatical. The non-hiding features play no role, since they only map to objects of simple types, which are grammatical in any case.

For Ex. 2, the only clause in $\mathcal{P}_h(\mathcal{G})$ will be

$$\mathit{gram}(X) \leftarrow X = (\mathit{ne\_list} \wedge \mathrm{hd} : Y_1 \wedge \mathrm{tl} : Y_2)\ \&\ \mathit{gram}(Y_1)\ \&\ \mathit{gram}(Y_2).$$

For constrained types, the definition is exactly parallel, except that additionally we need to add the information from the implicational constraints.

Definition 21 **($\mathcal{P}_c(\mathcal{G})$).** Let $\mathcal{G}$ be a grammar and C the set of constrained types in $\mathcal{G}_{\mathrm{IC}}$. For each type $t \in C$, define

$$Clause(t) := gram(X) \leftarrow X = (t \wedge \mathrm{IC}(t) \wedge f_1{:}\,Y_1 \wedge \ldots \wedge f_n{:}\,Y_n) \;\&\; gram(Y_1) \;\&\ldots\&\; gram(Y_n),$$

where $f_1 \ldots f_n$ are the hiding features of t and $X, Y_1, \ldots, Y_n$ are pairwise distinct variables not occuring in $\mathrm{IC}(t)$. Then let

$$\mathcal{P}_c(\mathcal{G}) = \{Clause(t) \mid t \in C \wedge t \in \mathcal{V}\}$$

For Ex. 2, the only clause in $\mathcal{P}_c(\mathcal{G})$ is

$$\begin{aligned} gram(X) \leftarrow X = \; & append \wedge \\ & ((\; \text{arg1: } e\_list \wedge \text{arg2: } L \wedge \text{arg3: } L \wedge \text{junk: } e\_list \\ & \;\vee \text{arg1: (hd: } H \wedge \text{tl: } T_1) \wedge \text{arg2: } L \wedge \text{arg3: (hd: } H \wedge \text{tl: } T_2) \wedge \\ & \quad \text{junk: (hd: } (append \wedge \text{arg1: } T_1 \wedge \text{arg2: } L \wedge \text{arg3: } T_2) \wedge \\ & \qquad \text{tl: } e\_list)) \wedge \\ & \;\text{arg1: } Y_1 \wedge \text{arg2: } Y_2 \wedge \text{arg3: } Y_3 \wedge \text{junk: } Y_4) \;\& \\ & gram(Y_1) \;\&\; gram(Y_2) \;\&\; gram(Y_3) \;\&\; gram(Y_4) \end{aligned}$$

The complete program is simply the union of the three sets we've just defined.

Definition 22 ($\mathcal{P}(\mathcal{G})$). $\mathcal{P}(\mathcal{G}) := \mathcal{P}_s(\mathcal{G}) \cup \mathcal{P}_h(\mathcal{G}) \cup \mathcal{P}_c(\mathcal{G})$

4.1 Soundness

By soundness of the translation we mean the right to left direction of the correct translation as defined in Def. 12.

Proposition 23 (soundness). *Let $\mathcal{G}$ be a grammar and ϕ a term. If the goal $gram(X)$ & $X = \phi$ has a $\mathcal{P}(\mathcal{G})$-answer then $\mathcal{G}$ admits ϕ.*

This result can be shown using the construction of a minimal model in Theorem 4.4. of [8]. The central result is Prop. 25.

Definition 24. Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation. If $\mathcal{U}' \subseteq \mathcal{U}$, then $\mathcal{I}_{|\mathcal{U}'} = \langle \mathcal{U}', \mathcal{S}_{|\mathcal{U}'}, \mathcal{A}_{|\mathcal{U}'} \rangle$ where for each $f \in \mathcal{F}$, for each $u \in \mathcal{U}'$, $\mathcal{A}_{|\mathcal{U}'}(f)(u)\downarrow$ iff $\mathcal{A}(f)(u)\downarrow$ and $\mathcal{A}(f)(u) \in \mathcal{U}'$, and if $\mathcal{A}_{|\mathcal{U}'}(f)(u)\downarrow$, then $\mathcal{A}_{|\mathcal{U}'}(f)(u) = \mathcal{A}(f)(u)$.

Proposition 25. *If $\mathcal{G}$ is a grammar, $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ a $\mathcal{FL}$-interpretation and $\mathcal{M}$ the minimal model of $\mathcal{P}(\mathcal{G})$ extending $\mathcal{I}$, then $\mathcal{I}_{|\{u \in \mathcal{U} \mid \langle u \rangle \in [\![gram]\!]^{\mathcal{M}}\}}$ is a model of $\mathcal{G}$.*

Proof sketch. The proof goes by induction on the construction of the minimal model of $\mathcal{P}(\mathcal{G})$. By case analysis for the clauses for simple, hiding and constrained types, we determine that at each step of the model construction, only objects licensed by the grammar are added to the *gram* relation. N.B. that the fact that we've embedded $\mathcal{FL}$ in the CLP scheme enables us to work directly with models.

Notice that Prop. 25 is much stronger than Prop. 23 requires. It shows that the solutions computed from $\mathcal{P}(\mathcal{G})$ have the persistence property. Suppose we want to know if our grammar admits term ϕ, submit the goal $gram(X)$ & $X = \phi$ and get $X = \phi'$ as an answer. Then we know that the grammar admits ϕ' and furthermore it admits any ϕ'' that entails ϕ'.

4.2 Completeness

If the translation were complete, we should be able to prove the following proposition, which we can't.

Proposition 26 (false). *Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation, $\mathcal{G}$ a grammar, $\mathcal{P}(\mathcal{G})$ the program associated with $\mathcal{G}$ and $\mathcal{M}$ the minimal model of $\mathcal{P}(\mathcal{G})$ extending $\mathcal{I}$. $\mathcal{U} = \{v \mid \langle v \rangle \in [\![gram]\!]^{\mathcal{M}}\}$.*

Let us consider an example that exhibits the incompleteness.

Example 3. Suppose we add the following implication to our earlier example (Ex. 2).

$$person \Rightarrow \text{father: gender: } male$$

This would be translated to

$$gram(X) \leftarrow X = (person \wedge \text{father: gender: } male \wedge \text{father: } Y_1) \ \& \ gram(Y_1)$$

which goes into an infinite loop on the query

$$gram(X) \ \& \ X = person$$

as well as the cyclic query

$$gram(X) \ \& \ X = (person \wedge \text{father: } X).$$

We can only provide finite proofs. This means that we can provide solutions only for terms that are satisfiable in models with certain properties. A good point to start looking, obviously, is the class of grammars for which the parsing problem (more generally, the admissibility problem) is decidable.

Definition 27 (finite model property). A grammar $\mathcal{G}$ has the finite model property[7] just in case for each ϕ, if $\mathcal{G}$ admits ϕ, then there is a model $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ of $\mathcal{G}$ with finite $\mathcal{U}$ s.t. $[\![\phi]\!]^{\mathcal{I}} \neq \emptyset$.

[7] The name was suggested by Mark Johnson (p.c.).

This restriction is similar to the off-line parsability constraint of LFG [10]. To see if a given grammar has the finite model property is clearly undecidable. However, given our discussion in the introduction, we get the following result.

Proposition 28. *If a grammar $\mathcal{G}$ has the finite model property, then the admissibility problem for $\mathcal{G}$ is decidable.*

One might expect that our translation can handle all grammars in this class correctly, since it is certainly possible to give such a translation. However, such a translation would have to be indirect and very inefficient. The problem arises from cyclic solutions – the search for those would have to be explicitly coded by the translation. Recall the grammar in Ex. 3. It has the finite model property, yet any finite model with an object of type *person* would have to be cyclic. The search for such cyclic models can not be directly coded in a logic program, one would have to use a meta-interpretation approach – not talk about the objects directly but about some datastructure that encodes objects. A translation that correctly handles all grammars with the finite model property would thus result in inefficient programs. We prefer to further restrict the class of grammars that we translate correctly (and efficiently).

Definition 29 ($Path_{\mathcal{I}}$). If $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ is an interpretation and $u \in \mathcal{U}$, then $Path_{\mathcal{I}}(u) = \{\pi \mid \mathcal{A}^*(\pi) \text{ is defined }\}$

Proposition 30. *Let $\mathcal{I} = \langle \mathcal{U}, \mathcal{S}, \mathcal{A} \rangle$ be an interpretation, $\mathcal{G}$ a grammar, $\mathcal{P}(\mathcal{G})$ the program defined for $\mathcal{G}$ and $\mathcal{M}$ the minimal model of $\mathcal{P}(\mathcal{G})$ extending $\mathcal{I}$.*

If for each $u \in \mathcal{U}$, $Path_{\mathcal{I}}(u)$ is finite, then for each $u \in \mathcal{U}$, $u \in \{v \mid \langle v \rangle \in [\![gram]\!]^{\mathcal{M}}\}$.

Proof. The proof runs by induction on $|Path_{\mathcal{I}}(u)|$. □

This boils down to the finite model property with an acyclicity condition added. Let us recapitulate the results of this section. We started out by saying that our translation doesn't work correctly in all cases, and indeed cannot do so for theoretical reasons. We then restricted our attention to grammars for which the admissibility problem is decidable. This is no restriction for linguistic applications, since Pollard and Sag [15] actually require HPSG grammars to be decidable. They don't say which problem they want to be decidable, but we may assume that it's the admissibility problem.

A large subclass of the decidable grammars are those with the finite model property, which may be viewed as a generalisation of the off-line parsability constraint. To obtain completeness, we put an acyclicity condition on top of the finite model property. This might be a bit strict in view of the fact that some HPSG grammars do use cyclic structures. However, this is the only option if we want to keep our translation as direct and simple as it is. As we argued above, extending our approach to the full class of finite model grammars would result in programs that are much less efficient. This, by the way, is true of whatever approach we use for solving the admissibility problem. It is not a problem specific to translating to logic programs.

There is a completely different approach to obtaining completeness that we haven't considered yet. Instead of restricting the class of grammars, we could also use a different semantics for logic programs. Instead of taking a least fixpoint (lfp) approach to program semantics, as is usual and as [8] did, we might consider using a greatest fixpoint (gfp) semantics.[8] If we did this, the completeness problem would vanish, we conjecture that our translation is complete with respect to a gfp semantics of logic programs. This is not so surprising since we've turned the parsing problem for logic programs from an entailment problem into something closer to a satisfiability problem.[9] However, this is only a theoretical solution, since we're interested in the results concerning the procedural semantics of logic programs under a lfp semantics. In fact, we're interested in practical systems that we can use to run the results of our translation.

5 Conclusion

We have presented a compilation scheme that translates HPSG constraint grammars into efficient logic programs by using a simple classification of types. The compilation is sound and for a large class of grammars, it is complete. It is also optimal in two senses. First, it is as complete as it can be without becoming too inefficient. Secondly, it is as efficient as it can be without becoming unsound.

Several versions of the compiler have been implemented. One version compiles directly into Prolog programs. This suffers from non-termination problems for anything but toy grammars due to Prolog's left-to-right depth-first search strategy. Good results have been obtained using a compilation into the CLP language CUF [7], which offers a more refined search strategy and the declaration of delay patterns as procedural annotations.

5.1 Other approaches

We are not aware of any other work that solves the admissibility problem for HPSG constraint grammars via a translation to logic programs. However, there are approaches to solving related problems by directly providing a procedural semantics, not via a translation. Bob Carpenter [4] shows how to do this for constraint grammars under open world type inference. An interesting difference is that he requires his models to be finite in the first place (which puts his approach outside first-order logic). Consequently, he gets different completeness

[8] I would like to thank Bill Rounds for pointing this out to me.

[9] It is a satisfiability problem if we consider the completion of the program [5].

results – since the finite structures are enumerable, his approach is complete for solving the admissibilty problem (with some restrictions having to do, as in our case, with cyclic structures).

Another approach that also addresses a similar problem is that of [2]. They also define the admissibility problem as a first order satisfiability problem and give a proof system that is complete for detecting inadmissibility. Although the system does not allow disjunction, it could be easily extended in that direction. Their work is also unique in that it uses a lazy evaluation technique, which improves termination properties.

There are two reasons why one might prefer the translation based approach over the direct approaches mentioned above:

1. The resulting programs actually run on existing architectures, no special system needs to be implemented.
2. The resulting programs can be optimised in general or for specific tasks (such as parsing or generation) using known program transformation techniques (see [14] for recent results in program transformation from a natural language processing perspective). This is a very important point, since in natural language processing, one is generally interested in solving specific problems (e.g., the parsing problem) for a given grammar, and the information provided by specific knowledge about the query can guide the search for a solution.

References

1. Aït-Kaci, H. (1984). *A lattice theoretic approach to computation based on a calculus of partially ordered type structures.* PhD thesis, University of Pennsylvania.
2. Aït-Kaci, H., Podelski, A., and Goldstein, S. C. (1993). Order-sorted theory unification. Technical Report 32, Digital Equipment Corporation.
3. Boolos, G. S. and Jeffrey, R. C. (1974). *Computability and Logic.* Cambridge University Press, third edition.
4. Carpenter, B. (1992). *The logic of typed feature structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press.
5. Clark, K. L. (1978). Negation as failure. In Gallaire, H. and Minker, J., editors, *Logic and Databases*, pages 293–322. Plenum Press, New York.
6. Dörre, J. (1994). Feature-Logik und Semiunifikation. Arbeitspapiere des SFB 340 Nr. 48, Universität Stuttgart.
7. Dörre, J. and Dorna, M. (1993). CUF - a formalism for linguistic knowledge representation. In Dörre, J., editor, *Computational aspects of constraint based linguistic descriptions I*, pages 1–22. DYANA-2 Deliverable R1.2.A, Universität Stuttgart.
8. Höhfeld, M. and Smolka, G. (1988). Definite relations over constraint languages. Technical Report 53, LILOG, IBM Deutschland.
9. Johnson, M. (1994). Two ways of formalizing grammar. *Linguistics & Philosophy*, 17:221–248.

10. Kaplan, R. and Bresnan, J. (1982). The mental representation of grammatical relations. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press.
11. King, P. J. (1989). *A logical formalism for head-driven phrase structure grammar.* PhD thesis, University of Manchester.
12. King, P. J. (1994). An expanded logical formalism for head-driven phrase structure grammar. Arbeitspapiere des SFB 340 Nr. 59, Universität Tübingen.
13. Lloyd, J. W. (1984). *Foundations of Logic Programming.* Springer-Verlag, Berlin, Germany.
14. Minnen, G., Gerdemann, D., and Hinrichs, E. (1996). Direct Automated Inversion of Logic Grammars. *New Generation Computing 14(2).*
15. Pollard, C. and Sag, I. A. (1994). *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago.
16. Smolka, G. (1992). Feature-constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87.