Inherently Reversible Grammars, Logic Programming and Computability

Marc Dymetman
CCRIT, Communications Canada
1575 boulevard Chomedey
Laval (Québec) H7V 2X2, Canada
dymetman@ccrit.doc.ca

ABSTRACT

This paper attempts to clarify two distinct notions of "reversibility": (i) *Uniformity of implementation* of parsing and generation, and (ii) reversibility as an *inherent (or intrinsic) property of grammars*. On the one hand, we explain why grammars specified as definite programs (or the various related "unification grammars") lead to uniformity of implementation. On the other hand, we define different intrinsic reversibility properties for such grammars—the most important being *finite reversibility*, which says that both parsing and generation are *finitely enumerable* (see text)— and give examples and counter-examples of grammars which possess or do not possess these intrinsic properties. We also show that, under a certain "moderation" condition on linguistic description, finite enumerability of parsing is equivalent to finite enumerability of generation.

1 Introduction

From the linguist's point of view, a grammar is a formal device which defines a recursively enumerable set of well-formed linguistic structures, each having, among other aspects, a phonological content (or, when dealing with written text, a string content) and a semantic content. Such a device is completely neutral as regards its uses for parsing (recovering semantic content from string content) or generation (recovering string content from semantic content).

From the computational linguist's point of view, on the other hand, the problem is how to implement such a grammar both as a parsing program and as a generation program, in such a way that these programs exactly reflect the content of the grammar. This we will call the *reversibility problem*.

Let us assume, for specificity, that the grammar has been presented as a definite program (a Prolog program).[1] Then the reversibility problem has a simple solution: use a *complete* interpreter for definite programs—for instance a top-down interpreter having a breadth-first search procedure[2]—and directly use the grammar as the program both for parsing and for generation. In the parsing mode, for any given string x, the program will enumerate all semantics $y_1, y_2, \ldots$ assigned to it by the grammar, and similarly, in the generation mode, for any given semantics y, the program will enumerate all semantics $x_1, x_2, \ldots$ assigned to it by the grammar. This is a striking property of definite programs: they are reversible in the sense that they naturally lead to uniformity of implementation of the parsing and generation modes (see §4).

So the reversibility problem is solved, and we can spend the next few years skimming through Fodor's (not Jerry's) guides in travel bookstores?

Not quite. First, the standard depth-first interpreter for definite programs is an incomplete one, and this problem must be circumvented in some way. Second, and more crucially, even when using a complete interpreter, parsing (and similarly generation) does not in general terminate: the program may well enumerate $y_1, y_2, \ldots$ ad infinitum. This is even true if, in fact, there are only a finite number of solutions $y_1, y_2, \ldots, y_k$, or even, in the extreme case, no solution at all: the program may not be "aware" that it has at some point already exhausted all the solutions that it will eventually

[1] We could have made some other choice, for instance some unification grammar formalism. The advantage of using definite programs in the present discussion is that they embody the whole unification paradigm in its purest form, that unification of terms is conceptually simpler (and less prone to misunderstandings) than unification of DAGs, and that the denotational and operational semantics of definite programs have been thoroughly studied.

[2] See e.g. [7, p. 59] and section §2.1.3. See also [19] in this volume for a related approach.

find, and go on eternally looking for new solutions.

The source of this problem can be more or less severe: It may simply be due to the grammar's implementation as a certain program, or it may be intrinsic to the grammar.

If it is not intrinsic to the grammar, one may attempt some kind of program transformation on the grammar—for instance a local transformation as goal reordering in clause bodies [4, 16], or a global transformation as left-recursion elimination [5, 3] [3] —in order to get a parsing program which displays a finite behavior.[4] If such a transformation is possible in principle, we say that, intrinsically, the grammar has a *finitely enumerable parsing problem.*[5] One example of a class of grammars which respect this crucial condition is provided by *offline-parsable* DCGs, once compiled as definite programs (see [9]).[6]

We have limited the former discussion to the case of parsing. The case of generation is treated in a parallel fashion, and one can similarly define the conditions in which a grammar is said to have an intrinsically *finitely enumerable generation problem.* When a grammar is such that it has a finitely enumerable parsing problem and a finitely enumerable generation problem, we call the grammar *inherently finitely reversible.*

When this is the case, it is by definition possible to find a program P_p for parsing and a (not necessarily identical) program P_g for generation such that, for any string x, P_p enumerates all associated semantics y and terminates, and, for any semantics y, P_g enumerates all associated strings x and terminates.

Inherent finite reversibility is the concept which, in my opinion, permit us to capture formally the intuitive notion that a certain grammar is, or is not, "reversible".

[3] Or more generally, any transformation exploiting theorems provable of the grammar. Another instance of this technique is provided by the addition of conservative guides in [5], which "strengthen" the grammar on the basis of properties inferable from its form.

[4] Another popular approach is to use a special-purpose interpreter, exploiting properties of the grammar known a priori. [18] and [14] use this approach in the case of generation (see below).

[5] The description is simplified; see §3 for the exact definition.

[6] See also [17] for a discussion of offline-parsability in the context of generation.

2 Definite programs and computation

2.1 Denotational and operational semantics of a definite program; complete and incomplete interpreters

A definite program P is a finite set of clauses of the form (*non-unit clauses*):

$$p(T_1, \ldots, T_n) \leftarrow$$
$$p_1(T_{11}, \ldots, T_{1n_1}) \cdots p_m(T_{m1}, \ldots, T_{mn_m})$$

or of the form (*unit clauses*):

$$p(T_1, \ldots, T_n) \leftarrow$$

where the the p, p_i are predicate symbols and the T_i, T_{ij} are terms over a certain Herbrand universe of ground terms H.

We will suppose that, among the predicates p defined by P, one, r, is privileged and plays the role of the "main predicate" in the program. We will assume that r is of arity one.[7]

2.1.1 Denotational semantics

The denotational, or declarative, semantics of program P can be defined as the least fixed point of a certain operator on Herbrand interpretations which we will not describe here (see [7]). Informally, the denotations of the predicate symbols p are defined as n-ary relations $p(x_1, \ldots, x_n)$ over H, built as the limit of a bottom-up process which starts from the unit clauses and uses the non-unit clauses to add new instances to each relation.

In particular, this process defines the unary relation $r(x)$ on H, which we shall call the *denotational semantics of the main predicate r relative to program P.*

Let T be a term over H; We define the *specialization of* $r(X)$ *on* T as the relation $r_T(x)$ on H defined by:

$$r_T(x) \stackrel{\text{def}}{\equiv} r(x) \wedge x \sqsubseteq T$$

where $\sqsubseteq$ is the relation of subsumption. In case the term T is a variable X, we say that X is the *trivial specialization*, and we note that the relation $r_X(x)$ is identical to the relation $r(x)$.

[7] This assumption permits to simplify the exposition, but is not otherwise necessary.

2.1.2 Operational semantics

While the denotational semantics of P is an intrinsic property of P, its operational semantics is defined relative to some *interpreter*.

For our purposes, we will informally define an interpreter as a computational mechanism:

$$intpr(P, r(T))$$

which is input a definite program P, as well as a query $?r(T)$—where r is P's main predicate and T a term over H—and which outputs a *finite or infinite* "list of answers":

$$T_1, T_2, \ldots, T_k, \ldots$$

The T_k's are terms over H, ground or not, whose ground instances provide the "solutions" to query $?r(T)$. If the list of answers is infinite, the interpreter will not stop; If it is finite the interpreter may or may not stop: if it does, we will say that the interpreter *terminates* on query $?r(T)$.

Consider now the relation r'_T on H defined by:

$$r'_T(x) \stackrel{\text{def}}{\equiv} x \sqsubseteq T_1 \vee x \sqsubseteq T_2 \vee \cdots \vee x \sqsubseteq T_k \vee \cdots$$

We say that r'_T is the *operational semantics* of the main predicate r of P, for *specialization* T, relative to interpreter *intpr*.

Keeping the same notations as above, consider now the denotational semantics $r(x)$ of r relative to P, and consider its specialization $r_T(x)$.

Interpreter *intpr* is said to be *sound* iff one has, for any P, r, T:

$$\forall x \in H \quad r'_T(x) \Rightarrow r_T(x);$$

and to be *complete* iff:

$$\forall x \in H \quad r_T(x) \Rightarrow r'_T(x).$$

Soundness is a minimal requirement for an interpreter, and we will always assume it, but completeness is a requirement which is not always met in practice.

2.1.3 Complete and incomplete interpreters

The "standard" interpreter for definite programs uses a top-down, depth-first search algorithm. It is sound but not complete. Its non-completeness is due to the fact that it is depth-first: if its search-tree contains infinite branches, the interpreter will be "caught" in the first one and will never explore the branches—maybe leading to success—to the right of this branch in the search-tree [7, pp. 59–60].

By contrast, a top-down, breadth-first interpreter, i.e. one which explores nondeterministic choices (between the different clauses competing for resolution of the same atomic goal) in parallel[8] is complete [7, pp. 59].

The naïve bottom-up interpreter, which in essence directly calculates the denotational semantics of P, and filters a posteriori the semantics $r(x)$ through the constraint that the solutions unify with T, is also a complete algorithm.

2.2 Computational behavior of a definite program relative to an interpreter

We now consider a program P, having r as main predicate, the denotation of r relative to P being the relation $r(x)$ on H. We also consider a specialization T, i.e. a term on H.

We will compare the denotational content of P to its computational behavior, and describe three possibilities: (i) P *enumerates* r *on* T, (ii) P *discovers* r *on* T, and (iii) P *finitely enumerates* r *on* T. The interpreter is supposed to be fixed beforehand.

We say that:

- P *enumerates* r on specialization T iff:

  $$\forall x \in H \quad r'_T(x) \iff r_T(x),$$

  in other words, iff its list of answers:

  $$T_1, T_2, \cdots T_k, \cdots$$

  exactly "covers" the denotational semantics r_T.[9]

- P *discovers* r on specialization T iff:

  1. P enumerates r on T;
  2. If r_T is the uniformly false relation on H, then P terminates on T.[10]

- P *finitely enumerates* r on specialization T iff:

  1. P enumerates r on T;

[8] Or, alternatively, uses a *fair* search rule, i.e. one which "shares its attention" among all paths in the search-tree.

[9] This will always be the case if the interpreter is sound and complete, as seen in §2.1.2.

[10] Therefore, when r_T is uniformly false, the list of answers is empty, *and* the program is "aware" of this fact (i.e. it terminates).

2. P terminates on T.[11]

We simply say that *P enumerates (discovers, finitely enumerates) r* iff P enumerates (discovers, finitely enumerates) r on the trivial specialization X.

We have the obvious entailments:

P finitely enumerates r on $T \Rightarrow$ P discovers r on $T \Rightarrow$ P enumerates r on T. (1)

It is often the case that one is interested in the computational properties of a given definite program relative to a certain *class* of specializations. For instance, when using a grammar—given as a definite program—for parsing, one will consider all queries where some of the variables are ground (the string to parse) and others (the semantic form) are not, and one will want to consider the computational properties of the program relative to this class of specializations. When using the definite program for generation, one will be interested in another class of specializations, and will want to consider the computational properties of the program relative to *that* class of specializations.

Let $\mathcal{S} = \{T\}$ be a set of (not necessarily ground) terms on H, indexed by a finite or infinite set I. We call $\mathcal{S}$ a *class of specializations*. We say that:

- *P enumerates r on $\mathcal{S}$* iff, for all $T \in \mathcal{S}$, P enumerates r on T;
- *P discovers r on $\mathcal{S}$* iff, for all $T \in \mathcal{S}$, P discovers r on T;
- *P finitely enumerates r on $\mathcal{S}$* iff, for all $T \in \mathcal{S}$, P finitely enumerates r on T.

The mutual entailments between these properties are similar to the ones given in (1).

2.3 Intrinsic computational properties of a definite program

Let $\mathcal{S}$ be a class of specializations, and let $r(x)$ be an arbitrary unary relation on H. We suppose here that programs are evaluated with respect to a sound and complete interpreter, which has been fixed once and for all, and we say that:

- r is *enumerable on $\mathcal{S}$* iff there exists a definite program P which enumerates r on $\mathcal{S}$.
- r is *discoverable on $\mathcal{S}$* iff there exists a definite program P which discovers r on $\mathcal{S}$.
- r is *finitely enumerable on $\mathcal{S}$* iff there exists a definite program P which finitely enumerates r on $\mathcal{S}$.

These three notions, taken together, constitute a *"computability hierarchy"* where enumerability is the weakest condition, discoverability is an intermediary condition, and finite enumerability is the strongest condition. These computability conditions can be described more intuitively in the following way:[12]

- r is enumerable on $\mathcal{S}$ if there exists a program P such that, for any $T \in \mathcal{S}$, P is able, given infinite time, to find terms $T_1, T_2, \cdots$ such that:

$$\forall x \in H \quad r(x) \wedge x \sqsubseteq T \iff x \sqsubseteq T_1 \vee x \sqsubseteq T_2 \vee \cdots$$

- r is discoverable on $\mathcal{S}$ if there exists a program P which is furthermore able to decide in finite time, for any $T \in \mathcal{S}$, if there actually exists an x such that:

$$r(x) \wedge x \sqsubseteq T$$

- r is finitely enumerable on $\mathcal{S}$ if there exists a program P which is furthermore able to find in finite time, for any $T \in \mathcal{S}$, terms $T_1, T_2, \cdots, T_k$ such that:

$$\forall x \in H \quad r(x) \wedge x \sqsubseteq T \iff x \sqsubseteq T_1 \vee x \sqsubseteq T_2 \cdots \vee x \sqsubseteq T_k$$

Let $\{X\}$ be the set having for only element the trivial specialization X; $\{X\}$ is called the *trivial class of specializations*. We will simply say that r is *enumerable* (resp. *discoverable*, *finitely enumerable*) iff r is *enumerable* (resp. *discoverable*, *finitely enumerable*) on the trivial class $\{X\}$.

Let $\mathcal{G} = H$ be the set of all ground terms of H. $\mathcal{G}$ is called the *class of ground specializations*. The following properties—which we will not prove here—establish links between the notions that we have just defined and the classical notions of recursively enumerable relations and recursive relations:

r is a recursively enumerable relation on H iff r is enumerable on the trivial class of specializations $\{X\}$; if this is the case, then for any class of specializations $\mathcal{S}$, r is enumerable on $\mathcal{S}$. (2)

[11] In particular, the relation r_T is, loosely speaking, "finitely representable as a union of terms $T_1, T_2, \cdots T_k$" *and* the program is "aware", at a certain point, that it has exhausted the possible answers.

[12] Note that these definitions critically depend on the relative scopes of quantifiers $\exists P \; \forall T \in \mathcal{S} \; \ldots$: it is essential that program P be the same for all specializations T in $\mathcal{S}$.

r is a recursive relation on H iff r is discoverable on the class of ground specializations $\mathcal{G}$ iff r is finitely enumerable on the class of ground specializations $\mathcal{G}$. (3)

3 Grammars and their computational uses

Let $X\#Y$ denote, in infix notation, the term $\#(X,Y)$. In the context of this paper, we take a *grammar* to be a definite program G having as its main predicate the unary predicate r, and we will assume that the clauses defining r are of the form:

$$r(X\#Y) \leftarrow \cdots$$

X will be called the "p-parameter", Y the "g-parameter". Generally, the p-parameter will represent a character string, and the g-parameter a semantic form.[13]

3.1 Six computational problems

A grammar can be used either to *enumerate* well-formed structures or to check whether certain fully instantiated values of the parameters can be *accepted.* We distinguish six computational problems (grouped into four types) which can be solved with a grammar: *p-enumeration, p-acceptation, g-enumeration, g-acceptation, bi-enumeration, bi-acceptation.* These problems are defined, together with comments on their computational properties, using the terminology of §2.3. This permits us to characterize the different positions a given grammar can occupy on the "computability hierarchy" —enumerability/discoverability/finite enumerability—relatively to each of these problems.

3.1.1 p-enumeration and p-acceptation

The *p-enumeration problem* or *parsing problem* is the problem of enumerating, for any fixed ground term x, all ground terms y such that $r(x\#y)$. The *p-acceptation problem* or *decision problem for parsing* is the problem of checking, for any fixed ground term x, whether there exists a ground term y such that $r(x\#y)$ is true.

The same specialization class is associated with both these problems, namely the class $\mathcal{GP} = \{x\#Y\}_{x\in H}$ consisting in all the terms $x\#Y$ where x is any ground term, and Y is a certain variable (whose name is indifferent).

Let's consider in turn, with respect to $\mathcal{GP}$, the different positions the grammar—or equivalently, its denotational semantics r—can occupy on the computational hierarchy, from strongest to weakest:

Finite enumerability When r is finitely enumerable on $\mathcal{GP}$, it is *in theory* possible to find a program P such that, for any given (ground) value x of the p-parameter (the string), the program enumerates all the solutions to the parsing problem and terminates. These solutions are given implicitly as a finite list of answers $TY_1,\ldots,TY_k$: the TY_i's are terms whose ground instances y are the looked-for values of the g-parameter (the semantics associated with string x by the grammar). We also say that, with the grammar at hand, *p-enumeration is finitely enumerable*, or simply, that *parsing is finitely enumerable.* This is an inherent property of the grammar, and, in practice, this property does not necessarily entail that finding a program P to exploit will be obvious.[14] For instance, *offline-parsable grammars* [9] can be shown to possess a finitely enumerable parsing problem, but algorithms which are able to make use of this property are by no means trivial [9, 13, 3].[15]

Discoverability If r is not finitely enumerable on $\mathcal{GP}$, it may still be discoverable on $\mathcal{GP}$. By definition, this means that it is possible to find a program P such that, for any given (ground) value x of the p-parameter, if there is no value y of the g-parameter corresponding to x, then the program will "recognize" this fact in finite time and terminate with an empty list of answers; if, on the other hand, there are solutions y corresponding to x, then the program will enumerate them, but maybe not terminate. If this property holds, we also say that with the grammar at hand, *p-enumeration is discoverable*, or, simply, *parsing is discoverable.* One can easily prove (although we will not do it here) that this property is equivalent to the decidability (in the classical sense) of the p-acceptation problem. In other words:

[13] We thus take r to be a unary relation which "encodes" a binary relation. This is unessential, but permits us to use the concepts of the previous section, developed for unary relations, without having to generalize them to n-ary relations.

[14] See footnote 17.

[15] These papers do not use the concept (or, a fortiori, the terminology) "finite enumerability of parsing", which, to my knowledge, appears here for the first time (see however [6], for the related notion of "Universal Parsing Problem").

p-enumeration is discoverable if and only if p-acceptation is decidable.[16]

Enumerability By the definition of a grammar as being a recursively enumerable mechanism, and by property (2), r is enumerable on any specialization class, and in particular on $\mathcal{GP}$.

3.1.2 g-enumeration and g-acceptation

The *g-enumeration problem* or *generation problem* is the problem of enumerating, for any fixed ground term y, all ground terms x such that $r(x\#y)$. The *g-acceptation problem* or *decision problem for generation* is the problem of checking, for any fixed ground term y, whether there exists a ground term x such that $r(x\#y)$ is true.

The specialization class is associated with both these problems is the class $\mathcal{GG} = \{X\#y\}_{y\in H}$ consisting in all the terms $X\#y$ where y is any ground term, and X is a certain variable (whose name is indifferent).

The situation is exactly symmetrical to the case of p-enumeration and p-acceptation, and we can define, in the same way, the notions: *"generation is finitely enumerable"* and *"generation is discoverable"* (which is equivalent to *"g-acceptation is decidable"*).

3.1.3 bi-enumeration

The *bi-enumeration problem* is the problem of enumerating all ground terms x, y such that $r(x\#y)$.

The specialization class associated with this problem is the class $\mathcal{TRIV} = \{X\#Y\}$ which contains the single term $X\#Y$.

For non-degenerate grammars, it is not the case that r is finitely enumerable on $\mathcal{TRIV}$, for this would entail in particular that any string recognized by the grammar is subsumed under one of the terms in a fixed finite set of terms $T_1, \ldots, T_k$. This is a slightly weaker property than saying that there are finitely many strings recognized by the grammar, but is still a very unlikely property for a grammar.

On the other hand, by definition, r is enumerable on $\mathcal{TRIV}$. It can be shown easily that it is also discoverable on $\mathcal{TRIV}$.[17]

[16] An immediate consequence of this property (linking the p-enumeration problem with the p-acceptation problem) is the fact that a grammar which is finitely enumerable for parsing has a decidable p-acceptation problem. The converse is clearly false (see §6 for a counter-example).

[17] This is because: (i) In case the grammar generates nothing, there is a trivial program which, on query $?r(X\#Y)$, produces an empty list of answers and terminates and (ii) if this is not the case, then the grammar itself may serve as an enumerating program (perhaps a non-terminating one). Note that this does *not* entail that by looking at the grammar, one is actually able—even in principle—to decide which of these two situations actually holds! This is an extreme instance of the remark made above (in the discussion of finite enumerability of parsing) that the existence in principle of a program meeting certain criteria does not imply that it is obvious, or indeed possible, to find such a program.

3.1.4 bi-acceptation

The *bi-acceptation problem* is the problem of checking, for any fixed ground terms x and y, whether $r(x\#y)$ is true.

The specialization class associated with this problem is the set $\mathcal{G} = \{x\#y\}_{x,y\in H}$ of ground specializations.

It can be shown that r is finitely enumerable on $\mathcal{G}$ iff it is discoverable on $\mathcal{G}$ iff the relation r on H is recursive in the classical sense. When this is the case, one says that *bi-acceptation is decidable.*

Again, by property (2), r is enumerable on any specialization class, and in particular on $\mathcal{G}$.

REMARK. Suppose that parsing is finitely enumerable, that is, r is finitely enumerable on $\mathcal{GP}$. This obviously implies that r is also finitely enumerable on $\mathcal{G}$. Therefore, one has:

> parsing is finitely enumerable $\Rightarrow$ bi-acceptation is decidable;

and, by the same reasoning:

> generation is finitely enumerable $\Rightarrow$ bi-acceptation is decidable.

On the other hand, the weaker property that p-acceptation is decidable (or similarly, that g-acceptation is decidable) does not seem to entail that bi-acceptation is decidable.

4 Definite programs, uniformity of implementation, and reversibility

It is sometimes stated that various grammatical formalisms, based on a variant or another of unification, are "reversible". It should more properly be said that they are "well-adapted" to reversible grammar implementations. The paradigmatic case of a grammar given as a definite program G makes this especially clear.

We know, from the discussion of §3.1.1 and §3.1.2, that we always have: (i) r is enumerable

on $\mathcal{GP}$ and (ii) r is enumerable on $\mathcal{GG}$; we therefore know that there exist programs P_p and P_g which enumerate r respectively on $\mathcal{GP}$ and $\mathcal{GG}$. But in fact we have more: if we use a sound and complete interpreter, we can simply take $P_p = P_g = G$. This follows from the fact that, by definition, *relatively to such an interpreter*, G enumerates r_T, for *any* specialization T (see §2.1.1):

- G enumerates r on $\mathcal{GP}$;
- G enumerates r on $\mathcal{GG}$.

To be more concrete, suppose that we use a complete top-down interpreter; Its behavior will be along the following lines:

1. On query $?r(X\#Y)$, the interpreter returns the (generally infinite) list of answers

   $$T_1, T_2, \ldots, T_k, \ldots$$

   where each T_i is a term of the form $A_i\#B_i$; The (generally infinite) "union" of these terms "exactly covers" the query;

2. On a query of the form $?r(x\#Y)$, where x is a ground term, the interpreter returns the list of answers

   $$T_1 \sqcup (x\#Y), T_2 \sqcup (x\#Y), \ldots, T_k \sqcup (x\#Y), \ldots$$

   where $\sqcup$ is the operator of term unification, and where, with some abuse of notation, only the terms $T_i \sqcup (x\#Y)$ for which unification is possible actually appear in the list;

3. On a query of the form $?r(X\#y)$, where y is a ground term, the interpreter returns the list of answers

   $$T_1 \sqcup (X\#y), T_2 \sqcup (X\#y), \ldots, T_k \sqcup (X\#y), \ldots$$

   (with the same abuse of notation as above).

This is a rather striking property of definite programs: different "input modes" can be implemented *using one and the same interpreter and one and the same program.* (This property strongly contrasts with other programming paradigms, for instance functional or imperative ones. Programs of these types typically map an input x to an output y, and, while it is indeed true that, for a given y, the set of x_i which can serve as its input is recursively enumerable, the interpreter that could implement the (nondeterministic) mapping $y \mapsto x$ would have to be widely different from the "normal" interpreter for the language at hand.)

However, "reversibility" in this sense only means *uniformity of implementation* for different modes of use of a grammar. *Intrinsic finite reversibility* which is defined in the next section, gives a much stronger criterion of grammar reversibility.

5 Inherently reversible grammars

We say that a grammar G is (inherently) finitely reversible iff, in the terminology of §3.1.1 and §3.1.2, G is such that:

1. parsing is finitely enumerable;
2. generation is finitely enumerable.

In other words, G is finitely reversible iff there exists a program P_p for parsing and a (not necessarily identical) program P_g for generation such that, relative to some sound and complete interpreter:[18]

1. On a query of the form $?r(x\#Y)$, where x is any ground term, P_p returns a finite list of answers

   $$x\#T_1, x\#T_2, \ldots, x\#T_k$$

   and stops.

2. On a query of the form $?r(X\#y)$, where y is any ground term, P_g returns a finite list of answers

   $$T'_1\#y, T'_2\#y, \ldots, T'_l\#y$$

   and stops.

In order to guarantee that a grammar is finitely reversible, some strong assumptions must be made on its form. An example of such assumptions is provided by the class of *Lexical Grammars* described in [5].[19]

Lexical grammars are presented as definite programs. They all share the same core of rules, which describe basic compositionality assumptions (string compositionality, syntactic compositionality, semantic compositionality), but may have different lexicons, which contain all the more specific linguistic knowledge.

[18] In fact, one can also take here an incomplete interpreter such as the standard Prolog interpreter *stintpr*. Obviously, if programs P_p and P_g exist for a sound and complete interpreter *intpr*, one can also find such programs P'_p and P'_g relative to *stintpr*, by simulating *intpr* inside *stintpr*.

[19] See also [10] for a related approach.

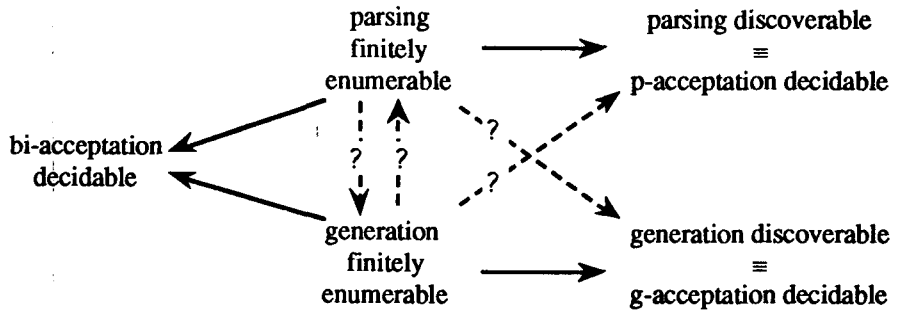

Figure 1: Computational problems associated with a grammar.

The hypotheses made on string compositionality in Lexical Grammars are simply that sister constituents concatenate their strings; they entail that parsing is finitely enumerable. The hypotheses on semantic compositionality are related to functional application and composition in categorial grammars (see e.g. [15]). They entail that generation is finitely enumerable.

A lexical grammar G is therefore finitely reversible. This does not imply that it can be used directly for parsing and for generation, but only, as seen previously, that there exist two programs P_p and P_g implementing G respectively for parsing and for generation. These programs are each obtained by a technique of adding to the grammar some redundant knowledge—respectively a *conservative guide for parsing* and a*conservative guide for generation*—and by applying a left-recursion elimination transformation (see [5]).

6 Some counter-examples to finite reversibility and a "moderation" condition on linguistic description

Fig.1 sums up graphically some of the relations which have been established in §3 between the computational problems associated with a grammar. The full arrows indicate entailments which have been established. The dotted arrows relate to a rather obvious question: What are the connections between the computational properties of parsing and those of generation? For instance, does the finite enumerability of parsing entail the finite enumerability of generation? If not, does it at least entail that g-acceptation is decidable? (The same questions can be asked in the reverse direction.) The answer is that, if no further assumptions are made (see below §6.3), then there are *no* connections. To show this, we now sketch one example which shows that finite enumerability of parsing does not even entail that g-acceptation is decidable.

6.1 A "grammar" related to Matiyasevich's theorem

Matiyasevich's theorem [2, p. 116] provides—among other things—a negative solution to Hilbert's tenth problem: "Does there exist an algorithm capable of solving all diophantine equations?", a diophantine equation being a multivariable polynomial in integer coefficients and whose variables range over **N**.[20]

Let K be a recursively enumerable, but non-recursive, subset of **N**. One corollary of Matiyasevich's theorem is the following property [2, p. 127–28]:

> There exists a polynomial $q(z_1, \ldots, z_n)$ in integer coefficients such that K is the set of values taken by q, for $z_1, \ldots, z_n$ ranging over all integers.

This corollary can be exploited to give an example of a "grammar" which has a finitely enumerable parsing problem, but such that its g-acceptation problem is not decidable.

Consider the relation $r(x\#y)$ which is true iff: (i) x is a string encoding any instance (for $z_1, \ldots, z_n$ ranging over the integers) of the expression $q(z_1, \ldots, z_n)$, using the symbols $0, \ldots, 9$, '+', '*', '(', ')', etc., and (ii) y is a term encoding the integer resulting from the arithmetical evaluation of $q(z_1, \ldots, z_n)$. This relation can easily be described

[20] The actual statement of Matiyasevich's theorem is stronger: "Every partially decidable predicate is diophantine" [2, p. 116].

by a "grammar" G: This grammar checks the well-formedness of string x, and calculates its "semantics" y.[21] G has the following properties:

- parsing is finitely enumerable: there is a program (namely G itself) finitely enumerating r on $\mathcal{GP}$. In effect, for any string x, this programs checks x for well-formedness and calculates the (single) "semantics" y resulting from the evaluation of x.
- g-acceptation is not decidable. Indeed, the problem of g-acceptation is the problem of deciding, for any given integer y, whether y is in the image of polynomial q, that is, whether y belongs to K. But K is a non-recursive set, hence the conclusion.

6.2 A "grammar" related to the undecidability of first-order logic

I will only *very* broadly sketch this example, which I think may provide useful insights on the importance of constraining "string compositionality" in a grammar.

Consider ordered pairs (x, y) of (ground) terms where x is a string encoding a certain first-order logic tautology, and y (the "semantics") is a derivation of x using a certain fixed set of axiom schemata and rules of inference for a complete system of first-order logic. Let's assume for simplicity that the given rules of inference always have two premises and one conclusion.[22]

A grammar G can be defined along the following general lines. The clauses of G correspond to the system's axiom schemata and rules of inference. Each clause corresponding to an axiom schema of name as defines "terminal constituents" $(x, as(x))$, where string x is any instance of schema as; each clause corresponding to an inference rule of name ir takes two "constituents" (x_1, y_1) and (x_2, y_2), and, if applicable (which is checked on the basis of strings x_1 and x_2), builds a new constituent (x, y), where x is the string obtained from x_1 and x_2 according to ir, and where y is a new derivation tree $ir(x, y_1, y_2)$. We have the following properties:

- generation is finitely enumerable: The generation problem is the problem, given a derivation tree y, of enumerating all formulas x that are associated with it. But y contains an explicit representation of x, so that generation is trivially finitely enumerable.
- p-acceptation is not decidable: The p-acceptation problem is the problem of checking if a string x can be derived from the axioms and the inference rules of the system. That is, it is the problem of checking if x is a tautology of first-order logic. By Church's undecidability result, this problem is undecidable.

6.3 Under a "moderation" condition on linguistic description, parsing is finitely enumerable iff generation is

The two counter-examples that we have just given have one property in common: the p-parameter can stay "small", while the g-parameter grows indefinitely "large", or conversely the g-parameter can stay small while the p-parameter grows indefinitely large. For instance, in the first counter-example, for a given value of y, there is no way to bound a priori the sizes of the integers $z_1, \ldots, z_n$ that may produce this y; in the second counter-example, there is similarly no way to bound a priori the sizes of proofs y for a given formula x.

In order to characterize this phenomenon formally, we will define a notion of "moderation" for a grammar G, defined as a definite program over the Herbrand universe H. As previously r is the unary relation representing the denotational semantics of G.

If a is a ground term in H, let us call *size* of this term, and denote by $size(a)$, the number of nodes in a. Grammar G will be called *moderate* iff there exist total recursive functions $f : \mathbf{N} \rightarrow \mathbf{N}$, and $g : \mathbf{N} \rightarrow \mathbf{N}$, such that:

$$\forall x, y \in H \quad r(x, y) \Rightarrow size(y) \leq f(size(x)) \\ \wedge\ size(x) \leq g(size(y)).$$

We have the following property:

If G is moderate, then, relative to G, parsing is finitely enumerable iff generation is finitely enumerable. (4)

Let us briefly sketch the proof: Suppose that parsing is finitely enumerable, then we know (see §3.1.4) that bi-acceptation is decidable. On the other hand, for any fixed ground term y, there are only finitely many ground terms x in H such that $size(x) \leq$

[21] This requires defining addition and multiplication of integers inside G, which presents no special problem.

[22] See for instance [8, p. 43–44] which describes a system having the two rules of inference $\frac{p \mid p \supset q}{q}$ and $\frac{p(x)}{\forall y p(y)}$ (where x is free in p). The second rule has one premise, but can easily be viewed as having two, if the premise $True$ is added to its original premise.

$g(size(y))$. Therefore, we can finitely enumerate all these x's, and for each of them, decide whether $r(x,y)$ holds. This shows that generation is finitely enumerable. The converse is proven in the same way.

Moderation might be claimed to be a "natural" constraint to impose on grammars used for "legitimate" linguistic purposes: One might want to argue that, in natural language, complexity of expression is a rather direct reflection of complexity of meaning. For example, semantic rules which reduced "you love him or you don't" to 'true', or "how much is 6 times 7 ?" to '?x.(x = 42)' would seem to be ruled out as valid linguistic descriptions. But we will not further pursue these tricky questions here.

Acknowledgments

Thanks to Pierre Isabelle, François Perrault, Patrick Saint-Dizier, Tomek Strzalkowski and Gertjan van Noord for their comments on an earlier version of this paper. English and content have bigly suffered from my lacking time to impose, as usual, its reading on Elliott Macklovitch.

Appendix

Examples of finitely reversible grammars that are inherently difficult to reverse

In this appendix, we give two examples of grammars that, although they are finitely reversible, are such that one mode is easy, while the reverse mode has a high degree of complexity. These examples are closely parallel, in the context of complexity, to the examples of section 6, which were concerned with computability.

Number products, cryptography and reversibility

Consider the binary relation $r(x\#y)$ which is true iff x is a string of the form:

$$N * M$$

where N and M are strings, interpreted as integers, of 0's and 1's and '$*$' is interpreted as multiplication, and where y is an integer equal to the product of M and N.[23] We impose a priori that integers M and N be strictly greater than 1.

This relation can be defined by a "grammar" G: this essentially simply involves constraining the "syntax" of x and defining multiplication by a set of definite clauses.

Implementing r in p-enumeration mode is easy: it involves verifying that x is well-formed, and computing its product according to specification G; In fact, G itself can be used for that purpose, using a standard interpreter.

On the other hand, *efficiently* implementing r in g-enumeration mode is extremely difficult, whatever the interpreter, program transformations, mathematical properties of prime factorization, ..., which are brought to the task. The fact that it is so difficult is the basis of the best known "public key cryptography" algorithm, RSA [11].

NP-complete problems and reversibility

A NP-complete problem is, informally, a problem for which solutions can be checked in polynomial time (relative to the length of the problem), but which requires more than polynomial time for the discovery of a solution [1].[24]

For specificity, let us focus on one NP-complete problem, namely the "3-colorability problem" which consists, given a certain graph x, in finding a coloring y for x using blue, green and red, in such a way that vertices sharing a common arc have different colors.

It is possible to state the problem as a definite program G, whose main relation is of the form $r(x\#y)$, x and y being suitable term encodings for the graph x and for the solution y. The solution y can be considered as implicitely containing a description of graph x.

It is obvious that *g-acceptation* is computationally easy (polynomial): it consists in verifying that the coloring y respects the coloring condition. On the other hand, *p-acceptation* is computationally costly: it consists in checking whether graph x has a solution, a problem which is at the present time believed to require exponential time.

[23] The string $N*M$ and the integer y are suitably encoded as ground terms on H.

[24] More exactly, which is *believed* to require more than polynomial time. This belief constitutes the content of the famous P≠NP conjecture.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] N. J. Cutland. *Computability*. Cambridge University Press, Cambridge, England, 1980.

[3] Marc Dymetman. A Generalized Greibach Normal Form for Definite Clause Grammars and the decidability of the offline-parsability problem, May 1991. Paper presented at the Second Meeting on the Mathematics of Language, Yorktown Heights, NY. (To be published).

[4] Marc Dymetman and Pierre Isabelle. Reversible logic grammars for machine translation. In *Proceedings of the Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Pittsburgh, PA, June 1988. Carnegie Mellon University.

[5] Marc Dymetman, Pierre Isabelle, and François Perrault. A symmetrical approach to parsing and generation. In *Proceedings of the 13th International Conference on Computational Linguistics*, volume 3, pages 90–96, Helsinki, August 1990.

[6] Mark Johnson. *Attribute-Value Logic and the Theory of Grammar*. CSLI lecture note No. 16. Center for the Study of Language and Information, Stanford, CA, 1988.

[7] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[8] Roger C. Lyndon. *Notes on Logic*. Van Nostrand, New York, NY, 1966.

[9] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics*, pages 137–144, MIT, Cambridge, MA, June 1983.

[10] François Perrault. Un nouveau formalisme de grammaire logique réversible. Master's thesis, McGill University, Montréal, Canada, 1991.

[11] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[12] Stuart M. Shieber. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 614–619, Budapest, August 1988.

[13] Stuart M. Shieber. Parsing and type inference for natural and computer languages. Technical note 460, SRI International, Menlo Park, CA, 1989. (Ph.D. dissertation, Department of Computer Science, Stanford University).

[14] Stuart M. Shieber, Gertjan van Noord, Robert Moore, and Fernando Pereira. A semantic-head-driven generation algorithm for unification-based formalisms. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 7–17, Vancouver, BC, Canada, June 1989.

[15] Mark Steedman. Dependency and coordination in the grammar of dutch and english. *Language*, 61(3):523–568, 1985.

[16] Tomek Strzalkowski and Ping Peng. Automated inversion of logic grammars for generation. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, pages 212–19, Pittsburgh, PA, June 1990.

[17] Gertjan van Noord. Towards convenient bidirectional grammar formalisms. In *Proceedings of the 13th International Conference on Computational Linguistics*, volume 2, pages 294–298, Helsinki, August 1990.

[18] Gertjan van Noord. BUG: A directed bottom-up generator for unification based formalisms. Technical report, RUU, Department of Linguistics, Utrecht, Holland, 1989.

[19] Remi Zajac. A uniform architecture for parsing, generation and transfer. In *Proceedings of the Workshop on Reversible Grammars in Natural Language Processing*, Berkeley, CA, 1991.