

Lexical Functional Grammar constraints and concurrent constraint programming

Peter Hancox

School of Computer Science, University of Birmingham, Birmingham, B15 2TT,
England

P. J. Hancox@cs.bham.ac.uk,

WWW home page: <http://www.cs.bham.ac.uk/~pjh/>

Abstract. Lexical Functional Grammar allows linguistic constraints to specify attributes and their values without using unification. The satisfaction algorithm for these constraints is within the generate-and-test paradigm and has the disadvantage of not being able to detect, at minimal cost, violations of the constraints as early as native speakers. Concurrent constraint languages, of which CHR is an example, allow searches to be incrementally constrained with goals delayed until they can be properly discharged. It is shown that linguistic constraints can be implemented in CHR to give early detection of satisfaction/violation of constraints, and also allows some further detection of redundancy and inconsistency.

1 Unification and constraints

Grammars used in syntactic parsers are, now, widely based on unification. The family of unification grammars has a broad membership from the computational: Prolog's Definite Clause Grammar (DCG); to the linguistic, for instance Head-Driven Phrase Structure Grammar and Lexical Functional Grammar (LFG). These formalisms typically use both syntactic and lexical categories (noun phrase, noun) and attribute/value pairs. The latter allow the expression of grammatical information, for instance the grammatical features of number, person, tense and so on.

Unification is used as the major information combining mechanism in these formalisms. Attributes and their values can be seen as collected into bundles and bundles can be unified to produce the attribute bundles of other linguistic objects. As an example, consider the combination of features for a determiner and noun which, together, give a feature bundle for a noun phrase:

$$\text{DET/the} + \text{NOUN/car} \rightarrow \text{NP/the car}$$
$$\begin{bmatrix} \text{PERS} & \text{3RD} \\ \text{SPEC} & \text{THE} \end{bmatrix} \quad \begin{bmatrix} \text{NUM} & \text{SING} \\ \text{PERS} & \text{3RD} \end{bmatrix} \quad \begin{bmatrix} \text{NUM} & \text{SING} \\ \text{PERS} & \text{3RD} \\ \text{SPEC} & \text{THE} \end{bmatrix}$$

The intimate association between attributes/values and unification has several implications. An obvious aspect is that it is relatively easy to impose a consistency requirement on feature bundles such that, for any feature, it shall have no more than one value. Thus, it is impossible for the number attribute to appear simultaneously with both the values of singular and plural. Second, the use of unification allows attributes to be referred to although, at the point of reference, they do not have a value. In programming language terms, it is the equivalent of accessing a variable, whether or not that variable has been set to a value.

The third consequence of unification is less obvious: it tends to lead the parser implementer into search algorithms of the generate-and-test variety. Classically, parsing with unification consists of generating partial structures, the validity of which are tested when they are unified. This is clearest in the most commonly used version of unification grammar, DCG, which, using Prolog's in-built top-down depth-first search, hypothesises (generates) structures which are tested and backtracked over when the test fails.

The work described here shows the weakness of the generate-and-test model of LFG constraints and demonstrates a new model of incremental constraints implemented in the concurrent programming language CHR. Not only does this model discharge constraints in an intuitively correct way but it also allows the detection of redundant and inconsistent constraints.

1.1 LFG and unification

LFG, while firmly in the unification grammar family, also uses alternative methods of specifying the values of attributes.

LFG uses two main representations of structure: c-structure is generated from the application of essentially context-free grammar rules; f-structure is a feature/attribute structure that allows values of attributes to be either atomic values (eg SING, 3RD, +) or f-structures. Thus, an f-structure may consist, in part, of embedded f-structures, although there are theoretical restrictions on the depth of embedding. As will be shown later, f-structures (including embedded f-structures) have important well-formedness conditions imposed upon them. While c-structure represents constituent structure in the same way as phrase structure in traditional syntax, f-structure represents the functional structure of sentences, setting it out as essentially a predicate with associated information (eg tense, mood, aspect) and the functions that the predicate governs (eg subject, object).

LFG specifies the production of c- and f-structures by schemata (equations) attached to context-free-like grammar rules and lexical entries. As an example, a NP could be specified in LFG as:

$$\begin{array}{c} \text{NP} \rightarrow \text{DET NOUN} \\ \uparrow = \downarrow \quad \uparrow = \downarrow \end{array}$$

the: DET, (\uparrow SPEC) = THE
 car: NOUN, (\uparrow NUM) = SING
 (\uparrow PRED) = 'CAR'

The $\uparrow = \downarrow$ schema states that the attributes and their values at the current node (eg DET or NOUN) should be unified with those of the higher node (NP).

It is simple to specify that the bundled attributes of a lower node should be the embedded value of a feature at the dominating node. Two typical rules from English are:

$$\begin{array}{l} S \rightarrow \quad NP \quad \quad VP \\ \quad \quad (\uparrow \text{ SUBJ}) = \downarrow \quad \uparrow = \downarrow \\ VP \rightarrow \quad \quad V \quad \quad NP \\ \quad \quad \quad \uparrow = \downarrow \quad (\uparrow \text{ OBJ}) = \downarrow \end{array}$$

In the first, the resulting f-structure will have a SUBJ attribute which will have as its instantiation the f-structure generated by the NP rule. In the second rule, there will be an embedded f-structure as the instantiation of the OBJ function. An f-structure can look like the following taken from [4]:

$$\left[\begin{array}{l} \text{SUBJ} \\ \text{PARTICIPLE} \\ \text{PRED} \\ \text{OBJ} \end{array} \left[\begin{array}{l} \left[\begin{array}{ll} \text{NUM} & \text{SING} \\ \text{PRED} & \text{'GIRL'} \\ \text{SPEC} & \text{A} \end{array} \right] \\ \text{PRESENT} \\ \text{'WASH}(\langle \uparrow \text{ SUBJ} \rangle (\uparrow \text{ OBJ})) \\ \left[\begin{array}{ll} \text{NUM} & \text{SING} \\ \text{PRED} & \text{'CAR'} \\ \text{SPEC} & \text{THE} \end{array} \right] \end{array} \right]$$

1.2 LFG and constraints

In addition to unification, LFG allows schemata to express constraints on the values attributes can take. These constraints are of two types: *constraining equations* that constrain the value an attribute can take and *existential constraints* that require a feature to be instantiated or uninstantiated. By the end of parsing, all constraints must be satisfiable.

Constraining equations It is possible to constrain a feature to be instantiated only to a specific value, while not actually causing a unification action:

$$(\uparrow \text{ SUBJ NUM}) =_c \text{ SING}$$

This states that the NUM feature embedded within the SUBJ feature bundle must be both instantiated and instantiated to SING. (It is possible to imagine a weaker version of this constraint: either the feature is instantiated to SING

or it remains uninstantiated.) Negative constraining equations are also available which, as their name suggests, require a feature either to have a value other than that given or to be uninstantiated:

$$\neg (\uparrow \text{SUBJ NUM}) =_c \text{SING}$$

The use of these constraints is often to enforce agreement without accidental unification. So, it is possible to specify that the NUM feature of an f-structure agrees with its subject's number feature, thus enforcing subject/verb agreement of English without, in the process, instantiating either feature:

$$(\uparrow \text{SUBJ NUM}) =_c (\uparrow \text{NUM})$$

Existential constraints These constraints specify that a named feature must be instantiated (without specifying what that value must be) or that a feature must remain uninstantiated.

1.3 Well-formedness conditions

LFG has three well-formedness conditions which it imposes on f-structures. As seen above, consistency requires that each feature has no more than one value. The other two, completeness and coherency, require a more detailed understanding of f-structure. Within each f-structure there should be an instantiated PRED feature. When this is contributed by a verbal lexical entity, this will include a specification of the governable functions that can occur with that verbal entity. The lexical entry used earlier:

$$\begin{aligned} \text{wash: V, } (\uparrow \text{ TENSE}) &= \text{PAST} \\ (\uparrow \text{ PRED}) &= \text{'WASH}((\uparrow \text{ SUBJ})(\uparrow \text{ OBJ}))' \end{aligned}$$

specifies that *wash* must occur with the two fully-formed grammatical functions of subject and object. This allows *Mary washed the car* but disallows **washed the car* because it is not complete: its lack of a subject leaves one governable function unsatisfied.

**Mary washed the car the sponge* is not coherent because it contains more governable functions than the PRED specification requires.¹

A governable function is held to be instantiated in an f-structure if its own, local PRED feature is itself instantiated (and is well-formed). So, returning to the example given above, it is complete because the f-structure's PRED feature's governable functions (SUBJ and OBJ) both have instantiated features. It is coherent because no governable functions other than those specified in the top-level PRED feature occur in the f-structure.²

¹ A comparison should be made with syntactically equivalent sentences such as: *Mary gave the baby the book*. It is possible to have sentences such as *Mary washed the car with a sponge* and *Mary washed the car with Jane* but the prepositional phrases act as adjuncts to the sentence.

² The set of governable functions is essentially defined by all those feature names that appear in PRED specifications.

The well-formedness conditions can be modelled using existential constraints. Completeness is modelled by adding an existential constraint on each governable function in the PRED specification. Coherency is modelled by adding a negative existential constraint on each governable function not given in the PRED specification. Assuming the existence of governable function OBJ2, the well-formedness example used above would be constrained as:

$(\uparrow \text{SUBJ PRED}) \wedge (\uparrow \text{OBJ PRED}) \wedge \neg (\uparrow \text{OBJ2 PRED})$
 (Alternatively, the last constraint could be written as $\neg (\uparrow \text{OBJ2})$.)

2 Testing constraints with generate-and-test

The original detailed description of LFG sets out a generate-and-test algorithm for constraints. The c-structure and f-structure are to be created with constraints collected. Once these structures are generated, constraints are tested for satisfaction along with testing for well-formedness. This approach has the virtues of being easy to explain and relatively easy to program.

Because of the family relationship between DCG and LFG, it is relatively straightforward to implement LFG in DCG. This suffices as a technique for experimentation and demonstration although, for practical purposes, Prolog has inherent disadvantages, for instance in its lack of structure sharing and inability to handle left-recursive rules. Here, DCG is used to model LFG's c-structure, with Prolog's term unification used to model attribute/value unification. LFG constraints, including well-formedness conditions, are collected to be solved in a post-parsing check.

Here, a small abstract grammar and lexicon is used as an illustration:

$\text{ntA} \rightarrow \text{ntB ntC}$ $\text{ntB} \rightarrow \text{ta tb}$ $\text{ntC} \rightarrow \text{ta ntB}$
 $\text{ta} \rightarrow \text{a}$ $\text{tb} \rightarrow \text{b}$ $\text{tc} \rightarrow \text{c}$

The first grammar rule can be implemented in DCG as:

```
ntA(ntA(NTB, NTC),
    fs(fA(FA), fB(FB), fC(FC), f1(F1), f2(F2), f3(F3), f4(F4)),
                                             Const0, Const) -->
    ntB(NTB, FA, Const0, Const1),
    ntC(NTC,
        fs(fA(FA), fB(FB), fC(FC), f1(F1), f2(F2), f3(F3), f4(F4)),
          Const1, Const).
```

and a lexical entry as:

```
ta(ta(a),
    fs(fA(_), fB(_), fC(_), f1(1), f2(2), f3(_), f4(_)), Const, Const)
    --> [a].
```

In a classic generate-and-test approach, constraints have to be collected, for instance in a difference list (in this example, in the variables `Const0`, `Const1`, `Const`) and then checked in a post-parsing test. The following (unrealistic) DCG clause shows how the various constraints could be coded:

```

ta(ta(a),
  fs(fA(_),fB(_),fC(_),f1(F1),f2(F2),f3(F3),f4(F4)),
  Const, [pos_const(F1, 1), neg_const(F2, 1), pos_exist(F3),
          neg_exist(F4)|Const])
--> [a].

```

(Such constraints could also be attached to grammar rules.) Well-formedness conditions can be implemented as outlined above:

```

ta(ta(b),
  fs(fA(fs(fA(_),fB(_),fC(_),f1(_),f2(FA2),f3(_),f4(_)),
    fB(fs(fA(_),fB(_),fC(_),f1(_),f2(FB2),f3(_),f4(_)),
    fC(fs(fA(_),fB(_),fC(_),f1(_),f2(FB2),f3(_),f4(_)),
    f1(_),f2(2),f3(_),f4(_)),
  Const, [pos_exist(FA2), pos_exist(FB2), neg_exist(FC2)|Const])
--> [b].

```

In the post-parse check, each member of the constraint list has to be checked to ensure it is satisfied, with code provided for testing each kind of constraint, eg:

```

test_constraint(pos_exist(Var)) :-
  nonvar(Var).

```

The generate-and-test technique doesn't seem intuitively correct and it is easy to see that some constraint violations that could be detected early are, in fact, detected late. It is possible to get an improvement in performance, most obviously by checking for constraint violations (and satisfactions) at the end of the processing of each non-terminal rule or, indeed, after processing each syntactic category. While the search tree would be pruned, there would be an increased cost in computation time. Such modifications don't change the essential generate-and-test inefficiencies of the problem.

Intuitively, the generate-and-test technique seems wrong. Suppose a constraint is used, as shown above, to impose subject/verb agreement. Then, for the sentence:

**a car are washed by the children*

the agreement constraint would not be applied and the ungrammaticality detected until the end of the sentence. However, a native speaker would detect the error as early as:

**a car are*

Looking at the problem from a programming perspective, it becomes a matter of when the satisfaction or violation of a constraint is detected. (The early detection of violations is particularly important as it triggers backtracking as soon as possible - in effect pruning the search tree as early as possible.)

3 Early detection of satisfactions and violations

Constraining equations are capable of early satisfaction and violation. For both negative and positive constraining equations, the testing of the constraint only need be delayed until the constrained feature is instantiated. In a strict interpretation of the positive constraint, it may be important that the constrained feature is not left uninstantiated: this is discussed below.

Existential constraints present a more complex problem. For a negative existential equation, violation can be detected as soon as the attribute it references is instantiated. If the attribute is left uninstantiated at the end of the parse, this is the equivalent of satisfaction. Note that there is no explicit satisfaction here, but this may not be significant in implementation. The satisfaction of positive existential constraints can be detected as soon as the attribute is instantiated. However, violations can't be detected until the whole input has been parsed and then only by some form of post-parsing check.

The strict interpretation of constraining equations becomes quite easy to model in this approach: it is formed of a weak constraining equation with existential equations added to ensure instantiation of attributes.

Thus, a successful implementation will be one where the evaluation of constraints is delayed, as far as possible, for only as long as attributes remain uninstantiated. The remainder of the paper presents and discusses an implementation in the concurrent constraint language, CHR. This allows incremental addition of constraints and also the manipulation of the constraint store which, in turn, allows for the detection of inconsistencies to an extent not envisaged in descriptions of LFG³.

4 CHR: a concurrent constraint language

Constraint Handling Rules (CHR)[3] is a general purpose constraint specification language that can be used in conjunction with several programming languages. In this investigation it is used with SICStus Prolog. CHR is sufficiently general that it is possible to implement the widely used constraint languages, eg CLP(B), CLP(\mathcal{R}) and CLP(FD).

CHR is a variety of concurrent constraint programming language where constraints are extended by adding dynamic scheduling. Constraints can be seen as processes which execute concurrently to the main program (in this case the DCG parser), communicating through the global constraint store[5]. The experimental implementation described below takes advantage of this to test for satisfaction and violation but also to introduce some pruning of the global constraint store and to detect logical inconsistencies.

³ Space limitations preclude a review of an early implementation in Prolog II[2] and an explanation of why Constraint Logic Programming over Finite Domains (CLP(FD))[1] does not provide the means to solve the problem.

4.1 LFG constraints in CHR

A concurrent constraint rule has the form:

```
Process :- Guard | Goal.
```

where *Process* is the process defined by the rule (and, unlike Prolog, this can consist of more than one “head”); *Goal* is the body of the rule (in the usual Prolog rule sense), and *Guard* is the delay condition that specifies when the rule can be fired.

When a CHR rule is first called, woken (by a variable being touched) or reconsidered (in backtracking), the guard is executed. Should the guard fail, then the next rule is tried or (if there are no alternative rules) the CHR rule is delayed until a variable is again touched.

If the guard succeeds, then the rule fires. For propagation rules (written with \Rightarrow) the body of the rule is executed without removing any constraint. Simplification rules (written using \Leftarrow) remove the matched constraint or constraints from the global constraint store and the body is executed. Simplification rules (written with \setminus) are a combination of propagation and simplification rules, such that any constraints preceding “ \setminus ” are kept in the global constraint store and those following are deleted.

The weak versions of the constraining equations are based on the use of the Prolog predicates *ground/1* and *var/1*. For the positive constraining equation the constraints are:

```
pos_const_weak(Attribute, Value) <=> ground(Attribute),
    ground(Value), Attribute == Value | true.
pos_const_weak(Attribute, Value) <=> ground(Attribute),
    ground(Value), Attribute \== Value | fail.
```

The negative constraining equations are similarly coded. Strict versions of these constraining equations are coded as a combination of weak constraining equations and existential constraints:

```
pos_const_strict(Attribute, Value) :-
    pos_const_weak(Attribute, Value),
    pos_exist(Attribute),
    pos_exist(Value).
```

Existential constraints are coded very simply:

```
pos_exist(Attribute) <=> ground(Attribute) | true.
neg_exist(Attribute) <=> ground(Attribute) | fail.
```

4.2 Discussion: satisfaction and violation of constraints

The concurrent nature of CHR allows constraints to be added incrementally during search, unlike the constrain-and-generate approach of CLP(FD). Incremental constraining exactly matches the “intuitive” view of LFG constraints set out above. A computation in CHR is not so much a tree search but a series of transitions between states. The posting of a constraint moves the program into a new state and the satisfaction or violation of a constraint is yet another state.

These constraints can be added to a DCG as embedded subgoals:

```
ta(ta(a),
    fs(fA(_),fB(_),fC(_),f1(_),f2(F2),f3(_),f4(_))) --> [a],
    { pos_const_weak(F2, 2) }.
```

In this implementation, LFG’s constraining equations are discharged as soon as their variables are instantiated (without the programmer having to pass messages or set flags). If the arguments are not instantiated, then the constraints remained in the store at the end of processing, without affecting the success of the parse. As has been seen above, the strict interpretation of the constraining equations is provided by simply adding existential constraints to the weak constraining equations.

The violation of negative existential constraints is detected as soon as the constraint’s variable is instantiated. Positive existential constraints present more of a challenge. If they are satisfied (ie their variable is instantiated) during parsing, they are immediately removed from the constraint store. However, if they are not satisfied then they are, necessarily, violated but this can only be detected by a post-parsing check⁴:

```
constraint_check :-
    findall_constraints(pos_exist(_), List_of_Pos_Exists),
    ground(List_of_Pos_Exists).
```

If compared with the model of LFG constraint processing set out in the objections to generate-and-test, it can be seen that the CHR version succeeds in implementing the detection of constraint satisfaction or violation at exactly the correct states in the parse process. As will be shown below, the use of incremental constraints programmed in CHR allows for the detection of inconsistent combinations of constraints.

4.3 Extensions to the CHR model

The power of CHR allows the specification of other rules that either prune the global constraint store or detect inconsistencies. Simplification rules can be used to remove duplicate constraints, as for the positive existential constraint:

⁴ As the strict versions of the constraining equations are programmed using existential constraints, they also rely on this post-processing check.

```
pos_exist(Attribute) \ pos_exist(Attribute) <=> true.
```

where the guardless rule will delete the second `pos_exist(Attribute)` from the store.

More significantly, some inconsistencies can be detected. The obvious contradictions are when a positive constraint occurs in the store with a corresponding negative constraint:

```
pos_const_weak(Attribute, Value), neg_const_weak(Attribute, Value)
    <=> fail.
pos_exist(Attribute), neg_exist(Attribute) <=> fail.
```

Less obvious is the inconsistency of a negative existential constraint on an argument of a positive constraining equation which, while not exactly a contradiction, seems inconsistent:

```
pos_const_weak(Attribute, Value), neg_exist(Attribute)
    <=> fail.
pos_const_weak(Attribute, Value), neg_exist(Value)
    <=> fail.
```

This is an extension to the processing of LFG constraints not explicitly set out in the detailed description of LFG.

5 Significance and conclusions

The significance of this work is that it provides an implementation of LFG's constraint system that, rather than following the standard generate-and-test description, uses concurrent constraint programming to implement an incremental constraint model. In so doing, it demonstrates that inconsistencies in LFG constraints can be detected early. While this application is specific and detailed, the principle of modelling with incremental constraints acting as agents has wider applicability, for instance in the modelling of reference in natural language as constraints waiting to be solved.

References

1. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. *Proceedings of the 9th International Symposium on Programming Languages: Southampton, 3-5 September 1997*. Berlin: Springer, 1997. 191–206.
2. Eisele, A.: A Lexical Functional Grammar system in Prolog. Department of Linguistics, Stuttgart University, 1984.
3. Frühwirth, T., Abdennadher, S.: *Essentials of constraint programming*. Berlin: Springer, 2003.
4. Kaplan, R. M., Bresnan, J.: Lexical-Functional Grammar: a formal system for grammatical representation. In: *The mental representation of grammatical relations*, edited by J. Bresnan. Cambridge, Mass: MIT Press, 1982. 173–281.
5. Marriott, K., Stuckey, P.J.: *Programming with constraints : an introduction*. Cambridge, Mass: MIT Press, 1998.