CHAPTER ONE

CONSTRAINTS AND LOGIC PROGRAMMING IN GRAMMARS AND LANGUAGE ANALYSIS

HENNING CHRISTIANSEN

1.1 INTRODUCTION

Constraints are important notions in grammars and languages analysis, and constraint programming techniques have been developed concurrently for solving a variety of complex problems. In this chapter we consider the synthesis of these branches into practical and effective methods for language analysis. With a tool such as Constraint Handling Rules, CHR, to be explained below, the grammar writer or programmer working with language analysis can define his or her own constraint solvers specifically tailored for the linguistic problems at hand. We concentrate on grammars and language analysis methods that combine constraints with logic grammars such as Definite Clause Grammars and CHR Grammars, and show also a direct relationship to abductive reasoning,

Section 1.2 reviews background on different but related notions of constraints in grammars and programming, and a a brief introduction to Constraint Handling Rules is given. The relation between abductive reasoning and constraint logic programming, most notably in the form of Prolog with CHR, is spelled out in section 1.3. Sections 1.4–1.5 shows hois materializes into methods for languages analysis together with Definite Clause Grammars and in the shape of CHR Grammars.

1.2 BACKGROUND

We assume a basic knowledge about the logic programming language Prolog and its grammar notation Definite Clause Grammars, DCG. There are

plenty of good standard books on logic programming and resources available on the internet, Christiansen (2010) gives a brief introduction intended for linguist students.

1.2.1 DIFFERENT NOTIONS OF CONSTRAINTS IN GRAMMARS AND LOGIC PROGRAMMING

The term constraints are being used with many, slightly different but overlapping meanings in computation and linguistics. Typically constraints $\mathscr{C}$ appear together with a generative model $\mathscr{G}$, where they serve as a filter reduce the extension of $\mathscr{G}$. For example, $\mathscr{G}$ may characterize a set of decorated syntax trees and $\mathscr{C}$ accepts only those trees whose decorations reflects correct inflection. In a Definite Clause Grammar, for example, the unification of attributes from different subtrees serve as constraints that limits the extension.

(Typed) feature structures can be seen as extensions to the standard terms of Prolog and considered more suited for modelling grammatical and semantic features of languages; they require a separately programmed unification algorithm when used with, say, Prolog-based grammars. There is a flourishing tradition for many sorts of such unification-based grammars or constraint-based grammars (where constraints typically are realized through a sort unification); we shall not go more details but refer to Pereira and Shieber (1987); Gazdar and Mellish (1989); Shieber (1992); Francez and Wintner (2012) for more information. Formally, these grammars are special cases of Knuth's attribute Grammars (1968).

However, attributes and constraints imposed on them are informative themselves and should not only be seen as devices to rule out syntactically wrong phrases or analyses thereof (or, strictly speaking, phrase-like structures). Syntactic features and semantic representations given as attributes may represent the desired the results of an analysis.

Another notion of constraints comes in from the tradition of constraint programming, in which a mathematical or logical problem concerns the assignment of values to variables that satisfies a number of constraints specified by a collection of predicates, where unification or its special case of syntactic equality are just examples of such predicates. For a general overview of this field, see Apt (2003), and for an introduction to its incarnation in logic programming, Jaffar and Lassez (1987). As a simple example, we may consider a constraint problem in the variables x and y in the domain of integer numbers given as $x \leq y \wedge y \leq x$. This constraint problem has an infinite

number of solution characterized by the reduced or solved constraint $x = y$. The device that performs this reduction is an algorithm called a constraint solver. In linguistic terms, we may consider a grammar for a language about mathematical entities, so that the analysis of two sentences produces the inequalities shown, one for each sentence, and we take $x = y$ as the meaning of the discourse consisting of those two sentences: we show this below in example 1.2.1 using the combined notations of DCG and Constraint Handling Rules.

Whereas as the aforementioned unification- or constraint-based grammars such as DCGs communicate meanings and features through the links and nodes of syntax trees, constraint logic programming allows to communicate through a global structure, typically called a constraint store. Thus an analysis may produce a syntax tree together with a constraint store in reduced form, by a cooperation between a parsing algorithm and the constraint solver. In case the constraint solver identifies a constraint violations, it may force the parser to try another decomposition strategy, perhaps leading to a rejection of the entire phrase being analyzed. While a constraint store in principle can be implemented as an attribute carried along the branches of a syntax tree, and thus provides no extension in a strict mathematical sense to the previous, the notion of a global store of information collected in an incremental way points in the direction of discourse analysis, i.e., the knowledge from one sentence is available for the analysis of the next one and so on, and the constraint solver serves as maintain constraint store, which new takes the role of a knowledge base.

1.2.2 CONSTRAINT HANDLING RULES, CHR

The programming language of Constraint Handling Rules (CHR) is an extension to Prolog that makes it possible to write specialized constraint solvers in a rule-base fashion. CHR consists of rewriting rules over constraint stores, and each time a new constraint is called, the given CHR program will accommodate it into the evolving constraint store (or perhaps produce a failure, leading to a shift of control in a driver process such as a parser). CHR is now available as an integrated part of several major Prolog systems. Here we give only a very brief introduction; a comprehensive account on CHR and its applications can be found in the book by Frühwirth (2009). CHR has three sorts of rules of the following forms.

Simplification rules:	$h_1,\ldots,h_n$	<=>	$Guard \mid b_1,\ldots,b_m$
Propagation rules:	$h_1,\ldots,h_n$	==>	$Guard \mid b_1,\ldots,b_m$
Simpagation rules:	$h_1,\ldots,h_k \setminus h_{k+1},\ldots h_n$	<=>	$Guard \mid b_1,\ldots,b_m$

The h's are head constraints and b's body constraints, and *Guard* is a guard condition (typically testing values of variables found in the head). A rule can be applied when its head constraints are matched simultaneously by constraints in the store and the guard is satisfied. For a simplification rule, the matched constraints are removed and the suitably instantiated versions of the body constraints are added. The other rules execute in a similar way, except that for propagation, the head constraint stays in the store, and for simpagation, only those following the backslash are removed. Prolog calls inside the body are executed in the usual way.

The indicated procedural semantics is consistent with a logical semantics based on a reading of simplifications as bi-implications, propagations as implications. A simpagation $H_1 \setminus H_2 <=> G \mid B$ is logically considered equivalent with the simplification $H_1, H_2 <=> G \mid H_1, B$, although it is executed in a different way.

Example 1.2.1 In section 1.2.1 we discussed informally a constraint solver for inequalities. In the following, we define a constraint solver with the described behaviour, using the constraint predicate `leq(`x`,`y`)` to represent $x \leq y$.

```
:- chr_constraint leq/2.
leq(A,B), leq(B,A) <=> A=B.
```

Solving inequalities is a standard introductory example for CHR; see Frühwirth (2009) for the few aditional rules necessary to handle all combinations of inequalities correctly. The CHR program can be tested from the Prolog system's command line as follows. A collection of constraints is given as a query, and the interpreter returns the result.

```
?- leq(X,Y), leq(Y,X).
X=Y?
```

In case the execution of a rule body leads to failure, it means that the constraints in the query are considered to be inconsistent with respect to the knowledge embedded in the current program. This may happen if a unification fails, e.g., `1=2`, or when the built-in predicate `fail` is called.

1.3 ABDUCTIVE REASONING IN LOGIC PROGRAMMING WITH CONSTRAINTS

Abductive reasoning was formulated as a principle by C.S.Peirce (1839–1914) as a third fundamental form to complement deduction and induction. Abduction has been studied in the context of logic programming, and it is also an important principle and a general metaphor for discourse analysis, so therefore it is relevant here to discuss this notion in a little details.

There are different formulations of what abduction means, but here we stay with the simplest form namely that abduction is the process of suggesting a suitable set of facts, which serves as a hypothesis, which, when added to our current knowledge, can explain an unexpected observation. Unexpected means here, not explainable from our current knowledge base without additional hypotheses added. Furthermore, the new hypothesis must not conflict with our current knowledge.

In symbols, when our current knowledge base is called $\mathscr{K}$ and the observation $\mathscr{O}$, we need to find a hypothesis or explanation $\mathscr{H}$ such that $\mathscr{K} \cup \mathscr{H} \Rightarrow \mathscr{O}$ but not $\mathscr{K} \cup \mathscr{H} \Rightarrow \mathit{false}$.

Discourse analysis can be understood as abduction as best known from the seminal paper by Hobbs *et al.* (1993): a listener *A* wants to figure out the new (to *A*) knowledge embedded in a discourse produced by a speaker *B*. For *A*, the discourse is an observation $\mathscr{O}$, and when he concludes, based on his current knowledge $\mathscr{K}$ "Aha, now I understand, *B* wants to convey the message or knowledge $\mathscr{H}$ – this makes sense and can explain why he is telling this story $\mathscr{O}$." This fits the general pattern of abduction shown above. One of the advantages of viewing discourse understanding as abduction – as opposed to a compositional assembly process from the bits of meanings embedded in each word – is that presupposed knowledge can be extracted: The utterance of the sentence "Now her husband is drunk again", if assumed to be true, can only be explained if 1) the husband is drunk, and 2) that this is a regularly recurring event. A more detailed analysis might also add (if it was not known), that 3) the husband suffers from alcoholism, and 4) has access to alcohol. If, furthermore, the general setting new to the listener, he may conclude further "hmmm, there are a female and a male character involved, and they happen to be married".

To understand abduction in logic programming, we may consider a situation where a person has a formalized a knowledge base in terms of a logic program, call it $\mathscr{P}$, and makes an observation $\mathscr{O}$. He wants to check if $\mathscr{O}$ is really the case according to $\mathscr{P}$ by asking the query ?– $\mathscr{O}$ – and receives

Prolog's laconic answer `no`. However, it may be the case that if the program is extended with a set of additional facts, which we call $\mathscr{H}$ to conform with the pattern above, that $\mathscr{O}$ will succeed in the extended program $\mathscr{P} \cup \mathscr{H}$; thus $\mathscr{H}$ may be a proposal for an abductive explanation. However, this argument may not be sufficient as knowledge about the general setting puts some restrictions on which combinations of facts that can co-exists. For example, if the problem is a detective problem, i.e., $\mathscr{O}$ is an observed crime, a hypothesis such as `did_it(the_ butler)` may be rejected, if this implies `was_at(misty_london, the_butler)` and it is a known fact that `was_at(sunny_brighton, the_butler)`. This conflicts with the world knowledge that a person cannot be two different places at the same time. Thus, to do abduction in logic programming, it maybe suggested to add an additional component of so-called integrity constraints $\mathscr{IC}$ to put restrictions on which sets of facts that are acceptable as explanations. To fit the general characterization of abduction above, we may set $\mathscr{K} = \mathscr{P} \cup \mathscr{IC}$.

An abductive logic program can be defined as a triplet $\langle \mathscr{P}, \mathscr{IC}, \mathscr{A} \rangle$ where $\mathscr{A}$ is a collection of predicates called abducibles, from which explanations can be composed, and $\mathscr{P}$ and $\mathscr{IC}$ as above. Abducible predicates are usually assumed not to occur in the head of any clause of $\mathscr{P}$.

Console *et al.* (1991) formalized in an elegant way how the execution of abductive logic programs can be understood as an extension to Prolog's standard goal-directed recursive descent style: instead of failing when it runs an abducible atom not mentioned in $\mathscr{P}$, it simply adds it to the growing explanation. A consistency check with respect to $\mathscr{IC}$ needs to be incorporated, ideally in an incremental way in order to reduce the search space. When Prolog is combined with CHR, this is exactly what CHR is doing: when a constraint is encountered during the execution of a query, it is added to the constraint store, and the constraint solver – i.e., the current set of rules – will apply, thus serving as an incremental test of consistency. This relationship between abduction and CHR was discovered by Abdennadher and Christiansen (2000) and its combination with Prolog as just described by Christiansen and Dahl (2005a). The idea is elucidated by the following example.

Example 1.3.1 (Christiansen, 2009) The following program consist of two Prolog rules describing different ways that someone can be happy, together with a CHR component that defines a constraint solver for three predicate which may be understood as abducible.

```
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).

:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.
professor(X), rich(X) ==> fail.
```

The single CHR rules formalizes the real world experience that the salary payed to professors do not make them rich. The following query asks for how a certain professor may become happy, is shown together with the answer produced by the combined Prolog and CHR interpreter.

```
?- happy(henning), professor(henning).
professor(henning),
has(henning,nice_students) ? ;
no
```

The presence the additional information in the query, that the person is a professor, triggers the CHR rule when the hypothesis `rich(henning)` is suggested by the Prolog program, thus rejecting this alternative. Above, the semicolon is typed by the user with the meaning of asking for alternative answers, and the system's `no` assures there are no more solutions than the one reported.

As appears, the implementation of abductive logic programming with CHR requires no program transformation or additional interpretation overhead. The approach may be summarized as translation of terminology from one computational domain to another.

Abductive logic programming	**Constraint logic programming with CHR**
Abductive logic programs	Prolog programs with a little bit of CHR
Abducible predicate	Constraint predicate
Integrity constraints	CHR Rules
Program rules	Program rules
Explanation	Final constraint store

The precise details for this equivalence are spelled out in Christiansen (2009).

Most other approaches to abductive logic programming are based on metainterpreters written in Prolog, see Denecker and Kakas (2002) for an overview and a later approach by Mancarella *et al.* (2009). Several of these can handle negation, which is not possible in our CHR-based approach described above, but our favours in terms of efficiency and the flexibility from

using an existing, fully instrumented programming system. The similarity between constraint programming and abduction was also observed in an early paper by Maim (1992) before the appearance of CHR.

1.4 USING CHR WITH DEFINITE CLAUSE GRAMMARS FOR DISCOURSE ANALYSIS

Definite Clause Grammars, DCG, play well together with CHR for discourse analysis as discussed very briefly in section 1.2.1. While, technically speaking, this way of using CHR to gradually collect the knowledge contained in a discourse is an application of abductive reasoning, its use with DCG appears to fairly easy to understand also for students without any knowledge about abduction. This approach to discourse analysis is demonstrated by a simple example adapted from Christiansen (2014b).

Example 1.4.1 We consider a constraint solver to be used for the analysis of stories about the students at small university at some fixed moment of time. The university has a number of rooms and other places, where students are allowed to come. A section of those are two lectures halls encoded in the program below as `lectHall1`, `lectHall2`; a reading room `rRoom`, student bar `bar`, a garden `garden`, etc. There are currently two courses going on, `programming` in `lectHall1` and `linguistics` in `lectHall2`. The following constraint predicates are introduced, `in(`s`,`r`)` indicating that student s is in room r, `attends(`s`,`c`)` that student s attends course c, `can_see(`s_1`,`s_2`)` that student s_1 can see students s_2, and finally `reading(`s`)` indicating that student s is reading. A student can only be in one room at a time, and reading can take place in any room but the lecture halls. A constraint solver for this can be expressed in CHR as follows; the constraint `diff(`x`,`y`)` is a standard device indicating that x and y must be different (easily defined in CHR; left out for reasons of space).

```
:- chr_constraint attends/2, in/2, reading/1.

attends(St, programming) ==> in(St, lectHall1).
attends(St, linguistics) ==> in(St, lectHall2).
in(St, R1) \ in(St, R2) <=> R1=R2.
reading(St) ==> in(St, R),
           diff(R, lectHall1), diff(R, lectHall2).
```

A rule body may provide alternative suggestions for explaining different observations, so for example for for student *x* to see student *y*, they must be in the same room or they may see each other througg a video call using skype. The two additional constraints and a rule are introduced to capture this as follows; the semicolon stands for Prolog's disjunction (which is implemented by backtracking).

```
:- chr_constraint can_see/2, skypes/2.

can_see(St1,St2) ==> in(St1,R), in(St2,R)
    ; skypes(St1,St2), in(St1,R1), in(St2,R2),
      diff(R1,R2).
```

The CHR declarations shown so far define a constraint solver that can be used together with any parsing algorithm in order to collect knowledge from a discourse. Here we show a DCG in which the constraints are called directly from within the grammar rules.

```
story --> [] ; s, ['.'], story.
s --> np(St1),  [sees],  np(St2), {can_see(St1,St2)}.
s --> np(St),   [is,at], np(C), {attends(St,C)}.
s --> np(St), [is,reading], {reading(St)}.
np(peter)    --> [peter].
np(mary)     --> [mary].
np(jane)     --> [jane].
np(programming)  --> [the,programming,course].
np(linguistics) --> [the,linguistics,course].
```

Consider now a query `phrase(story,[peter,⋯])` for the analysis of the text *Peter sees Mary. Peter sees Jane. Peter is at the programming course. Mary is at the programming course. Jane is reading.* It yields the following answer, i.e., the final constraint store when the sequence text has been traversed by the grammar.

```
attends(mary,programming)      can_see(peter,jane)
attends(peter,programming)     can_see(peter,mary)
in(jane,X)                     reading(jane)
in(mary,lectHall1)             skypes(peter,jane)
in(peter,lectHall1)            diff(X,lectHall1)
                               diff(lectHall2,X)
```

Referring to the discussion of discourse interpretation as special case of abduction, section 1.3 above, we may say that this constraint store is a knowledge base – or explanation – necessary for the discourse to be correctly

produced. The variable written as "X" stands for Jane's location which is not determined from the discourse; it is only known not to be one of the lecture halls.

The example above displays an incremental analysis of the text, in which the knowledge learned up to a certain point is available for the analysis of the next sentence. Furthermore, the use of constraint techniques delays choices that cannot be resolved currently, but may be resolved later when new knowledge is introduced.

As an example of this methodology, we refer to Christiansen *et al.* (2007), who applied DCG and CHR in a similar way to analyze use case text (use cases as applied in software development) for producing UML class diagrams. This involved an approach to pronoun resolution expressed as CHR rules based on which persons has been mentioned so far and in which distance from the pronoun under consideration. This paper uses CHR also to generalize properties mentioned for specific person into properties for the class to which this person belongs.

This combination of CHR with Prolog has been presented initially as an language called HYPROLOG (Christiansen and Dahl, 2005a) with special tools for declaration of abducible predicates, which in addition to the CHR constraint declarations shown above, also generates facilities of a weak form of negation of abducible predicates plus some other utilities. It includes also a notion of assumptions (Dahl *et al.*, 1997) that are very much like abducibles, but are explicitly produced and possibly consumed; declarations of assumptions are also compiled into CHR,

1.5 CHR GRAMMARS

A DCG as we demonstrated above together with CHR executes as a top-down parser that uses backtracking when examining different alternative parses. CHR itself can be used in a straightforward way for bottom-up parsing which, as is well known, is more robust of errors and less restrictive on the context-free backbone of the grammars that can be used (e.g. Aho *et al.*, 1988). Christiansen (2005) has introduced a grammar notation, CHR Grammars, that is compiled into CHR analogously to the way that DCGs are compiled into Prolog. CHR Grammars features different kinds of context dependent rules having an expressive power that goes far beyond DCGs and other logic programming based grammar systems. The following example demonstrates how CHR can be used for bottom-up parsing. Each nonterminal in the grammar is represented as a constraint predicate with

two additional attributes to indicate the location of each occurrence in the entire string to be analyzed.

Example 1.5.1 We consider a small language of which *"Peter likes Mary"* is a typical sentence. The languages has nonterminal symbols *np*, *verb* and *sentence*. The string shown can be encoded by the following set of CHR constraints.

```
token(0,1,peter), token(1,2,likes), token(2,3,mary)
```

Each single token or phrase recognized carries indices corresponding the point immediately before, respectively after, the token or phrase in the input string. The lexical part of the grammar that classifies each single token is given by the following rules.

```
token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==> verb(N0,N1).
```

The rule to recognize a sentence is a follows.

```
np(N0,N1), verb(N1,N2), np(N2,N3)
    ==> sentence(N0,N3).
```

In order to analyze a text, a set of token constraints as shown above is entered as a query, and the rules will apply as many times as possible, and for this example leaving the following final constraint store, that describes the recognized phrases, including all subphrases; we do not repeat the token constraints but they will also be present.

```
np(0,1)          verb(1,2)
np(2,3)          sentence(0,3)
```

The example may be varied using simplification rules instead, leading to the removal of intermediate nonterminals. However, when propagation rules are used in case of an ambiguous grammar, all the different possible parses will automatically be produced. Notice also that is straightforward to add additional attributes to each constraint (=nonterminal) symbol to hold other interesting syntactic, semantic or other properties associated with a phrase. The use of additional CHR constraints and rules for abductive interpretation as demonstrated with DCGs as shown in section 1.4 above can be used here, by calling constraints corresponding to abducibles in the right-hand sides of the rules.

CHR Grammars support a wide range of grammatical patterns that can be translated into conditions on the position indices. This includes gaps between subphrases or requirements that certain subphrases must be present immediately before or after, or in an arbitrary distance, from the symbols being matched. For example, to express that a nonterminal `a` followed immediately by nonterminal `b` can be reduced into an `ab` if followed by a `c` in a distance between zero and ten positions, can be expressed in the following way.

```
a(N0,N1), b(N1,N2), c(N3,_)
    ==>  N3 >= N2, N3 =< N2+10 | ab(N0,N2).
```

However, as these indices are tiresome to write and easy to get wrong, CHR Grammars offer a high-level notation, and the CHR Grammar compiler translates this into the right constraints, index variables and guards to form CHR rules as the one just shown. The CHR rule shown above can be written as the following CHR Grammar rule.

```
a, b /- 0...10, c ::> ab.
```

The symbol written as three dots is a pseudo-nonterminal that can be given with or without limits for the length of the substring that it spans. The material to the right of the "`/-`" symbol indicates what is called a left context, and there is a similar marker "`-\`" for indicating left context. Such gaps are highly relevant for biological sequence analysis, e.g., for gene finding and protein structure prediction.

Another of CHR Grammar's features is parallel matching, indicated by an operator "`$$`", so the following rule will recognize an `a` phrase as a `special_a` if it has a length between 10 and 20 and that a `b` has been recognized inside the substring spanned by `a`.

```
a $$ 10...20 $$ ...,b,...  ::>  special_a.
```

A detailed description of all options can be found in (Christiansen, 2005) and at the CHR Grammar website (Christiansen, 2002) that has a comprehensive users' guide, several example grammars and source code that runs under the SWI and SICStus Prolog systems. The CHR grammar systems includes the same utilities for abduction and assumptions as the HYPROLOG system explained at the end of section 1.4 above.

CHR Grammars have been used for biological sequence analysis by, among others, Bavarian and Dahl (2006), for modeling various phenomena in natural language, e.g., (Aguilar-Solis and Dahl, 2004; Dahl, 2004).

CHR without the CHR Grammar notation has been used for variants of the bottom-up parsing method for analysis of hieroglyph inscriptions (Hecksher *et al.*, 2002) and for analysis of time expressions in pre-tagged biographic texts (van de Camp and Christiansen, 2012). The latter used multiple indices that located each token according to its occurrence its position as well as which sentence in which paragraph and in which document. The Chinese Word Segmentation Problem was approach with CHR Grammars by Christiansen and Li (2011); see also paper by these authors in the present volume.

1.6 CONCLUSION

The use of constraints as devices in grammars and as programming tools have been discussed with an emphasis on the amalgamation into practical tools for language analysis. The programming language of Constraint Handling Rules, CHR, was demonstrated as a tool for representation and evaluation of semantic and other information being extracted from a text under analysis. It has been argued elsewhere Christiansen and Dahl (2005b) that this approach may provide an integration of the semantic and semantic aspects of language analysis. The relation to abductive reasoning was shown, in that a straightforward use of constraints together with Definite Clause Grammars and CHR Grammars is an instance of abduction.

We have shown only fairly simple examples, but it should be emphasized that constraint solvers that model very complex semantic domains can be written in CHR; see (Frühwirth, 2009) and the growing literature on applications of CHR. As an example of this, we may mention a solver written in CHR for handling different sorts of calendric expressions, including resolving relative time expressions (Christiansen, 2014a) from partial information.

BIBLIOGRAPHY

Abdennadher, S. and Christiansen, H. (2000). An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer).

Aguilar-Solis, D. and Dahl, V. (2004). Coordination revisited - a constraint handling rule approach. In C. Lemaître, C. A. R. García, and J. A. González, editors, *IBERAMIA*, volume 3315 of *Lecture Notes in Computer Science*, pages 315–324. Springer.

Aho, A. V., Sethi, R., and Ullman, J. D. (1988). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Bavarian, M. and Dahl, V. (2006). Constraint based methods for biological sequence analysis. *Journal of Universal Computing Science*, **12**(11), 1500–1520.

Christiansen, H. (2002). CHR Grammar web site; released 2002. http://www.ruc.dk/~henning/chrg.

Christiansen, H. (2005). CHR Grammars. *Theory and Practice of Logic Programming*, **5**(4-5), 467–501.

Christiansen, H. (2009). Executable specifications for hypothesis-based reasoning with Prolog and Constraint Handling Rules. *J. Applied Logic*, **7**(3), 341–362.

Christiansen, H. (2010). Logic programming for linguistics: a short introduction to prolog, and logic grammars with constraints as an easy way to syntax and semantics. *TRIANGLE*, **1**, 31–64.

Christiansen, H. (2014a). Constraint logic programming for resolution of relative time expressions. In A. Beckmann, E. Csuhaj-Varjú, and K. Meer, editors, *Computability in Europe 2014*, Lecture Notes in Computer Science. Springer. To appear.

Christiansen, H. (2014b). Constraint programming for context comprehension. In P. Brézillon and A. Gonzalez, editors, *Context in Computing*. To appear.

Christiansen, H. and Dahl, V. (2005a). HYPROLOG: A new logic programming language with assumptions and abduction. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer.

Christiansen, H. and Dahl, V. (2005b). Meaning in Context. In A. Dey, B. Kokinov, D. Leake, and R. Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111.

Christiansen, H. and Li, B. (2011). Approaching the chinese word segmentation problem with CHR grammars. In *CSLP 2011: Proc. 4th Intl. Workshop on Constraints and Language Processing*, volume 134 of *Roskilde University Computer Science Research Report*, pages 21–31.

Christiansen, H., Have, C. T., and Tveitane, K. (2007). Reasoning about use cases using logic grammars and constraints. In *CSLP '07: Proc. 4th Intl. Workshop on Constraints and Language Processing*, volume 113 of *Roskilde University Computer Science Research Report*, pages 40–52.

Console, L., Dupré, D. T., and Torasso, P. (1991). On the relationship between abduction and deduction. *Journal of Logic and Computation*, **1**(5), 661–690.

Dahl, V. (2004). An abductive treatment of long distance dependencies in chr. In H. Christiansen, P. R. Skadhauge, and J. Villadsen, editors, *CSLP*, volume 3438 of *Lecture Notes in Computer Science*, pages 17–31. Springer.

Dahl, V., Tarau, P., and Li, R. (1997). Assumption grammars for processing natural language. In *ICLP*, pages 256–270.

Denecker, M. and Kakas, A. C. (2002). Abduction in logic programming. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer.

Francez, N. and Wintner, S. (2012). *Unification grammars*. Cambridge University Press, New York, NY.

Frühwirth, T. (2009). *Constraint Handling Rules*. Cambridge University Press.

Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in Prolog: An Introduction to Computational Linguistics*. Addison-Wesley Publishing Co., Reading, Massachusetts.

Hecksher, T., Nielsen, S. T., and Pigeon, A. (2002). A CHRG model of the ancient Egyptian grammar. Unpublished student project report, Roskilde University, Denmark.

Hobbs, J. R., Stickel, M. E., Appelt, D. E., and Martin, P. A. (1993). Interpretation as abduction. *Artificial Intelligence*, **63**(1-2), 69–142.

Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *POPL, Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 111–119.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2), 127–145.

Maim, E. (1992). Abduction and constraint logic programming. In *ECAI*, pages 149–153.

Mancarella, P., Terreni, G., Sadri, F., Toni, F., and Endriss, U. (2009). The ciff proof procedure for abductive logic programming with constraints: Theory, implementation and experiments. *Theory and Practice of Logic Programming*, **9**(6), 691–750.

Pereira, F. C. N. and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis*, volume 10 of *CSLI Lecture Notes Series*. Center for the Study of Language and Information.

Shieber, S. M. (1992). *Constraint-Based Grammar Formalisms*. MIT Press.

van de Camp, M. and Christiansen, H. (2012). Resolving relative time expressions in Dutch text with Constraint Handling Rules. In D. Duchier and Y. Parmentier, editors, *CSLP*, volume 8114 of *Lecture Notes in Computer Science*, pages 166–177. Springer.