Rule-based constraint propagation: theory and applications
Brand, S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE (Digital Academic Repository)
Chapter 5

Incremental Rule Generation

5.1 Introduction

While constraint propagation rules capturing the desired propagation of one or some constraints can be devised manually, in doing so, several issues arise. Designing appropriate rules requires expertise; their correctness must be guaranteed, and for more complex constraints, it may not even realistically be possible. In response to these difficulties, the issue of an automatic generation of rule-based constraint propagation algorithms has received considerable attention in recent years. [Apt and Monfroy, 2001] considers the generation of membership rules; [Ringeissen and Monfroy, 2000] examines a parameterised variant of them. [Abdennadher and Rigotti, 2002, Abdennadher and Rigotti, 2004] deal with more general constraint propagation rules. The latter approaches aim particularly at the CHR language and also discuss methods to generate CHR simplification rules allowing the deletion of constraints from the constraint store (see Section 2.2.1). In [Dao et al., 2002], the issue of automatic generation of solvers based on indexicals [Codognette and Diaz, 1996] is examined.

Common to most of these approaches is their paradigm that is essentially generate-and-test. Successively, candidate rules for constraint propagation are enumerated. A rule candidate is kept if it passes the correctness test against the constraint definition. In the deviating method of [Ringeissen and Monfroy, 2000], a conclusion is derived from a candidate premise, which itself comes from a syntactic enumeration process, however. [Abdennadher and Frühwirth, 2003] examines how to merge solvers written in the CHR language. Due to the expressiveness of CHR, the main aspects are termination and confluence. Here we concern ourselves only with constraint propagation rules where these two properties are no issues, which lets us focus on the constraint propagation.

In contrast to the generate-and-test approaches, we explore the idea of rule generation by incrementally modifying previously constructed rule sets. The key feature is that the input to the solver generation algorithm is already a set of
rules. The generation process consists in transforming the rule set into one that possesses desirable properties with respect to the associated constraints, such as the ability to establish a local consistency. An explicit definition of the constraints, for instance extensionally as the set of solutions, is unnecessary. The rule set is processed according to declarative transformation steps, meta rules, leading to the introduction of new rules or removal of existing rules. A number of benefits arise from this approach:

- First, the description of rule generation as an incremental process provides a new perspective on the origins of rule-based constraint solvers. This helps us to better understand such solvers and their propagation.

- Second, incremental solver generation reuses previously constructed rule sets. It also potentially increases the level of constraint propagation.

- Third, the incremental method can also be used as a universal rule generation method, by accompanying it with a pre-process that turns a constraint definition not based on rules into a set of simple initial rules.

While we first discuss incremental rule generation in general, the main part of this chapter deals with a specific type of rule, the membership rules. Our motivation for this focus is, on the one hand, that few useful statements can be made without fixing a specific language of constraint propagation rule (we elaborate on this issue below), and on the other hand, the relevance of membership rules.

We examine a variety of cases of incrementally generating sets of membership rules. In a justified sense, the central case is constructing a rule set \( R(C_1 \land C_2) \) for the conjunctive constraint \( C_1 \land C_2 \) from the rule sets \( R(C_1) \) and \( R(C_2) \) of its constituent constraints. The simple union \( R(C_1) \cup R(C_2) \) generally does not maximally propagate the conjunction \( C_1 \land C_2 \). Take the following rules, for example.

\[
\begin{align*}
C_1, x \in \{1 \} & \rightarrow C' \\
C_2, x \in \{2 \} & \rightarrow C' \\
C_1 \land C_2, x \in \{1, 2 \} & \rightarrow C' \\
\end{align*}
\]

In presence of the conjunctive constraints \( C_1 \land C_2 \) and \( x \in \{1, 2 \} \), none of the rules \( r_1, r_2 \) of the constituent constraints \( C_1, C_2 \) lets us obtain \( C' \). This shows why we would like to derive stronger rules such as \( r_3 \).

Furthermore, we discuss the cases of existential and universal quantification. If a constraint \( C \) is on a variable \( x \) then both \( \exists x . C \) and \( \forall x . C \) are constraints on the remaining variables of \( C \), and we explain how to construct the rule sets \( R(\exists x . C) \) and \( R(\forall x . C) \) based on \( R(C) \). We also discuss the auxiliary cases of extending the scope of a constraint to a new variable, and of extending the underlying domain by a new element, which means adding certain new solutions to the
5.2. Transforming Sets of Constraint Propagation Rules

A method of obtaining rules for a constraint from its extensional definition, alternatively as a set of solutions or non-solutions, makes incremental membership rule generation as capable as competing membership rule generation methods.

In all instances of membership rule generation, we focus on their most relevant feature, namely the relation to generalised arc-consistency.

5.2 Transforming Sets of Constraint Propagation Rules

A transformation of a rule set is a sequence of atomic steps introducing or removing single rules. We describe the admissible steps by meta rules with side conditions, applied to sets of constraint propagation rules. We write

\[
\frac{R}{R \cup \{r\}} \quad \text{(introduce)} \quad \frac{R}{R \setminus \{r\}} \quad \text{(remove)}
\]

where \( R \) is a rule set and \( r \) is a rule.

We consider two meta rules: subsumption, which deletes a rule, and derivation, which introduces a rule, based on the given rules.

5.2.1 Subsumption

Subsumption is a special case of redundancy of a rule with respect to a rule set, for the purpose of computing common fixpoints; see Section 4.3.2. We restrict ourselves here to a simple case and consider only propagation rules with the same body.

As a meta rule, we have

\[
\frac{R \cup \{A \rightarrow C, B \rightarrow C\}}{R \cup \{A \rightarrow C\}} \quad \text{if } (5.1) \quad \text{(gen-subsume)}
\]

where the constraints in \( A, B \) are on the variables \( X \) with domains \( D \), and the side condition is

\[
\text{Sol}(\langle A, X, D \rangle) \supseteq \text{Sol}(\langle B, X, D \rangle)
\]

(5.1)

Recall that \( \text{Sol}(\mathcal{P}) \) is the set of solutions of the CSP \( \mathcal{P} \), so (5.1) expresses that \( A \) is implied by \( B \).

We say that a rule is subsumed by a set of rules if it is subsumed by some rule in the set. So \( x < y \rightarrow C \) is subsumed by \( R \cup \{x \leq y \rightarrow C\} \).
5.2.2 Derivation

Two rules with identical body give rise to a new rule if the disjunction of their conditions, or something more restrictive, can be expressed in the underlying constraint language. Formally,

\[
\frac{R \cup \{A_1 \rightarrow C, A_2 \rightarrow C\}}{R \cup \{A_1 \rightarrow C, A_2 \rightarrow C, B \rightarrow C\}} \quad \text{if (5.2) and (5.3)} \quad \text{(gen-derive)}
\]

where the constraints in \(A_1, A_2, B\) are on the variables \(X\) with domains \(D\), and the side condition is

\[
\text{Sol}((B, X, D)) \subseteq \text{Sol}((A_1, X, D)) \cup \text{Sol}((A_2, X, D)). \quad (5.2)
\]

The idea of this transformation step is to compose the ancestor rules, at best into a descendant that in turn subsumes one or both ancestors. It is not difficult to show that correctness is preserved: if each rule in the original rule set is correct then so is each rule in the obtained rule set. However, note that the respective common fixpoints of the rule sets generally change!

While not needed for preservation of correctness, it is useful to require additionally

\[
\text{Sol}((B, X, D)) \nsubseteq \text{Sol}((A_i, X, D)) \quad \text{for } i = 1 \text{ and } i = 2 \quad (5.3)
\]

as otherwise the descendant rule would simply be subsumed.

5.2.1 Example. Suppose we know that the two rules

\[
x \neq y, y \neq z, z \leq w \rightarrow C, \\
x \neq z, z > w \rightarrow C
\]

are correct. Also let us assume that \textbf{alldifferent}, a constraint requiring pair-wise difference of its variables, is in the constraint language. By \textbf{(gen-derive)} we obtain the new rule

\[
\text{alldifferent}(x, y, z) \rightarrow C,
\]

whose condition does not imply, or is implied by, those of the ancestor rules. □

Generally, several possible candidates for the derived condition \(B\) in \textbf{(gen-derive)} exist, and they depend on the constraint language. Ideally, a suitable \(B\) can be constructed directly from \(A_1, A_2\). This is the case for membership rules.
5.3 Translating Sets of Membership Rules

We specialise the generic meta rules here for the language of membership rules. We refine the meta rules for subsumption and derivation, which allows us to characterise in terms of local consistencies a membership rule set closed under these meta rules.

While according to the definition of a membership rule the body of a rule can consist of multiple inequality constraints, for the purpose of this chapter we can assume that such a rule is decomposed into several membership rules with a single body constraints. So membership rules here are constraint propagation rules in the form

\[ C(x_1, \ldots, x_n, y), \ x_1 \in S_1, \ldots, x_n \in S_n \rightarrow y \neq a, \]

where each \( S_i \) is a set of constants, and \( a \) is a constant. In the following, \( C \), the constraint associated with the rule, is often irrelevant or clear from the context, and we omit it then from the notation. With the understanding that

\[ X = x_1, \ldots, x_n \quad \text{and} \quad S = S_1 \times \cdots \times S_n \]

we write the above membership rule concisely as

\[ X \in S \rightarrow y \neq a. \]

We proceed by specialising the subsumption and derivation transformations. Subsequently, we describe the rule set resulting from a stabilising derivation of such transformations. We show that if certain conditions on the source rule set are met then the resulting rule set enforces GAC.

5.3.1 Subsumption

A membership rule can be removed when another one performing the same domain reduction but with wider bounds on the variables is available:

\[
\frac{R \cup \{X \in S \rightarrow y \neq a, \ X \in P \rightarrow y \neq a\}}{R \cup \{X \in S \rightarrow y \neq a\}} \quad \text{if} \quad P \subseteq S \quad \text{(subsume)}
\]

It is easy to see that (subsume) is an instance of (gen-subsume): if \( P \subseteq S \), then every solution of the constraints \( X \in P \) is a solution of the constraints \( X \in S \).

5.3.1 Example. \( x \in \{2\} \rightarrow y \neq 1 \) is subsumed by \( x \in \{2,3\} \rightarrow y \neq 1. \) \( \square \)
5.3.2 Derivation

Two membership rules can sometimes be combined to form another one that allows the same domain reduction in new situations.

\[
R \cup \{X \in S \rightarrow y \neq a, X \in P \rightarrow y \neq a\} = R \cup \{X \in S \rightarrow y \neq a, X \in P \rightarrow y \neq a, X \in Q \rightarrow y \neq a\} \quad \text{if (5.4)}
\]

\[
\text{(derive}_k\text{)}
\]

\[
\begin{align*}
(a) & \quad Q_i = S_i \cap P_i \quad \text{for all } i \in [1..n], i \neq k, \\
(b) & \quad Q_k = S_k \cup P_k, \\
(c) & \quad Q_k \supset S_k \text{ and } Q_k \supset P_k, \\
(d) & \quad Q_i \neq \emptyset \quad \text{for all } i \in [1..n].
\end{align*}
\]

These four side conditions guarantee that the derived rule

- inherits correctness from its ancestor rules, (5.4.a) and (5.4.b), where one notices that every solution of \( X \in Q \) is a solution of \( X \in S \) or \( X \in P \) (compare with (gen-derive)),

- is not subsumed by any ancestor rule, (5.4.c),

- is a valid membership rule, (5.4.d).

It is useful to point out how we have used the constraint language underlying the membership rules. The disjunctive constraint "\( x_k \in S_k \) or \( x_k \in P_k \)" can directly be represented in this language, namely as \( x_k \in Q_k \).

A (\text{derive}_k) step depends on \( k \), and for two ancestor rules there may be several appropriate indices \( k \), satisfying (5.4). Note however, that no derived rule subsumes another with a different \( k \). In the following, when \( k \) is not relevant we write just (\text{derive}) instead of (\text{derive}_k).

5.3.2. Example. From

\[
\begin{align*}
x_1 \in \{1,2\}, \quad x_2 \in \{1,3\} & \quad \rightarrow \quad y \neq 2, \\
x_1 \in \{2,3\}, \quad x_2 \in \{2,3\} & \quad \rightarrow \quad y \neq 2
\end{align*}
\]

we obtain

\[
\begin{align*}
x_1 \in \{1,2,3\}, \quad x_2 \in \{3\} & \quad \rightarrow \quad y \neq 2 \quad \text{with } k = 1, \\
x_1 \in \{2\}, \quad x_2 \in \{1,2,3\} & \quad \rightarrow \quad y \neq 2 \quad \text{with } k = 2
\end{align*}
\]

by (\text{derive}_k). □
5.3.3 Result of the Meta Rule Closure

We examine now the properties of exhaustive applications, i.e., closures, of the meta rules (derive), (subsume) for membership rule sets. We proceed in two steps. First, we link the source rule set to the meta rule closure with respect to all correct membership rules associated with the constraint. Subsequently, we characterise the constraint propagation that a closed rule set achieves.

Atomic Rules

5.3.3. Definition. The membership rule \( C(X, y), X \in S \rightarrow y \neq a \) is atomic if each \( S_i \) is a singleton set.

The following note establishes an important 1–1 correspondence between an atomic rule and a non-solution of its associated constraint.

5.3.4. Note. The atomic rule \( C(X, y), X \in S \rightarrow y \neq a \) in which the variables \( X, y \) have the domains \( \mathcal{D} \) is correct if and only if the tuple \( d \in \mathcal{D} \) with \( \{d[X]\} = S \) and \( d[y] = a \) is not a solution of \( C \).

5.3.5. Example. The tuple \((1, 1, 0)\) is not a solution of the constraint \( \text{and}(x, y, z) \) expressing the conjunction \( x \land y = z \). It corresponds to the correct atomic rules

\[
\begin{align*}
\text{and}(x, y, z), & \quad x \in \{1\}, \; y \in \{1\} \quad \Rightarrow \quad z \neq 0, \\
\text{and}(x, y, z), & \quad x \in \{1\}, \; z \in \{0\} \quad \Rightarrow \quad y \neq 1, \\
\text{and}(x, y, z), & \quad y \in \{1\}, \; z \in \{0\} \quad \Rightarrow \quad x \neq 1.
\end{align*}
\]

We denote by \( \text{closure}(R) \) the rule set that results from applying (derive), (subsume) exhaustively. Here is the first observation: all ‘interesting’ rules are obtained by computing the closure of all atomic rules.

5.3.6. Lemma. Let \( R \) be a set of membership rules, all associated with the constraint \( C \). If \( R \) subsumes every atomic membership rule correct for \( C \) then \( \text{closure}(R) \) subsumes every membership rule correct for \( C \).

Proof. We argue by contradiction: Let us say that \( r = (X \in S \rightarrow y \neq a) \) is correct for \( C \) but not subsumed by \( \text{closure}(R) \). Without loss of generality we can assume that \( r \) is a most specific such rule, in the sense that all other rules \( X \in S' \rightarrow y \neq a \) with \( S' \subset S \), i.e., subsumed by \( r \), are also subsumed by \( \text{closure}(R) \).

Observe first that \( r \) is not atomic. Take then from the condition \( X \in S \) some \( S_k \) that is not a singleton, and partition it into \( S_k = P_k \cup Q_k \) where neither \( P_k \) nor \( Q_k \) is empty. Construct new rule conditions \( X \in P, X \in Q \) by just defining \( P_i = Q_i = S_i \) at the remaining indices \( i \neq k \).
Since \( r \) is a correct rule, so are the rules \( X \in P \rightarrow y \neq a \) and \( X \in Q \rightarrow y \neq a \). Furthermore, both these rules are strictly subsumed by \( r \), which means they are also subsumed by \( \text{closure}(R) \).

Thus, for each of the two rules, a subsuming rule contained in \( \text{closure}(R) \) exists. Enter these two subsuming rules into \( (\text{derive}) \). The resulting new rule must subsume \( r \), which contradicts our assumption.

With regard to \( (\text{subsume}) \), we remark that subsumption is a transitive relation. Therefore, if a rule is subsumed by a rule set then this is still the case after an application of \( (\text{subsume}) \) to the set.

We know now which 'seed rules' are necessary so that after closure there are rules for all correct propagations. Next, we establish the local consistency notion achieved by these propagations.

**5.3.7. Lemma.** Let \( R \) be a set of membership rules correct for their associated constraint \( C \). Let \( R \) subsume every rule correct for \( C \). Then the constraint \( C \) is closed under \( R \) if and only if \( C \) is generalised arc-consistent.

**Proof.** For the 'if' direction, suppose that the constraint \( C \) is closed under \( R \) but not under some correct rule \( r \notin R \). We show that \( C \) is not generalised arc-consistent. Let \( r = (X \in S \rightarrow y \neq a) \). We have thus \( C[X] \subseteq S \) and \( a \in C[y] \).

Since \( r \) is a correct rule, we know that for all \( d \) in the product of the variable domains we have that \( d[X] \in S \) implies \( d[y] \neq a \). The counter position is that \( d[y] = a \) implies \( d[X] \notin S \), and in turn \( d[X] \notin C[X] \), for all \( d \).

In other words, the partial instantiation \( \{y \rightarrow a]\) can not be extended to a solution of \( C \), so \( C \) is not GAC.

For the reverse direction, suppose that \( \{y \rightarrow a\} \) can not be extended to a solution of the constraint \( C \). So no \( d \) exist with \( d[y] = a \) and \( d[X] \in C[X] \). Then \( x_1 \in C[x_1], \ldots, x_n \in C[x_n] \rightarrow y \neq a \) is a correct rule; and as such is subsumed by \( R \). The subsuming rule in \( R \), however, is applicable to \( C \).

**From Atomic Rules to GAC**

**5.3.8. Definition.** Let \( R \) be a set of correct membership rules all associated with a constraint \( C \). \( R \) is called **atomically complete** with respect to \( C \) if \( R \) contains or subsumes every correct atomic rule associated with \( C \).

Here we have the important consequence of Lemmas 5.3.6 and 5.3.7.

**5.3.9. Theorem.** Assume that \( R \) is atomically complete w. r. t. a constraint \( C \) and let \( R_{cl} = \text{closure}(R) \). \( R_{cl} \) is sufficient for enforcing GAC on \( C \); that is, the constraint \( C \) is generalised arc-consistent if and only if \( C \) is closed under \( R_{cl} \).
5.3.4 Infeasible Rules

It is useful to characterise a rule by the following property of its condition.

5.3.10. Definition. A constraint propagation rule is called feasible if its condition is satisfiable:

\[ \mathcal{A} \rightarrow \mathcal{C} \quad \text{is feasible if} \quad \text{Sol}(\mathcal{A}, X, \mathcal{D}) \neq \emptyset, \]

where the constraints in \( \mathcal{A} \) are on the variables \( X \) with domains \( \mathcal{D} \).

For membership rules we find

\[ C(X, y), X \in S \rightarrow y \neq a \quad \text{is feasible exactly if} \quad S \cap C[X] \neq \emptyset. \]

Note that an infeasible rule is trivially correct. It is also often redundant.

The notion of membership rules stems from [Apt and Monfroy, 2001] where also the first algorithm for automatically generating such rules is described. We call this generation algorithm RGA here and revisit it in Section 5.6.2.

Since a rule set generated by RGA does not contain infeasible rules but does suffice to establish GAC, one may suspect that infeasible rules are without use. This is not the case, as we see now.

5.3.11. Example. The closure-based approach to membership rule generation, unlike RGA, may yield infeasible rules. It may also generate 'partially infeasible' rules. Define the constraint \( C \) on the variables \( x, y \) with domain \( \{1, 2, 3\} \) as in the following table. The RGA algorithm generates the GAC-enforcing rules \( R = \{r_1, \ldots, r_4\} \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( y \in {3} \rightarrow x \neq 1 )</th>
<th>( y \in {1, 2, 3} \rightarrow x \neq 2 )</th>
<th>( x \in {1, 2, 3} \rightarrow y \neq 2 )</th>
<th>( x \in {1} \rightarrow y \neq 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( r_1 )</td>
<td>( r_2 )</td>
<td>( r_3 )</td>
<td>( r_4 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( R \) is a minimal rule set in the sense of the strong redundancy notion in Chapter 4. Consider now the rule

\[ y \in \{2\} \rightarrow x \neq 1. \]

It is correct, but infeasible and redundant with respect to \( R \). While \( R \) is closed under (derive), (subsume), we can, however, apply (derive) to \( R \cup \{r_5\} \), and obtain

\[ y \in \{2, 3\} \rightarrow x \neq 1, \]

\( r_6 \)
which subsumes both \( r_1 \) and \( r_6 \). Completing the meta rule closure, we obtain the rule set \( R' = R - \{ r_1 \} \cup \{ r_6 \} \), which is minimal with respect to redundancy and enforces GAC, just as the original \( R \). However, for constraint propagation, the rule \( r_6 \) is preferable to \( r_1 \) since its condition is weaker, and \( R' \) is therefore preferable to \( R \).

Including infeasible rules in rule generation by the closure method lets us obtain rules that are more useful for constraint propagation. It is also generally unavoidable for completeness, Theorem 5.3.9, since some atomic rules may be infeasible (as \( r_6 \) above).

### 5.4 Cases of Incremental Rule Generation

We discuss now various useful instances of incremental rule generation, based on the meta rule closure. In each case, we assume that some source constraints \( C_1, \ldots, C_m \) with associated rule sets \( R_1, \ldots, R_m \) are given. We explain how a new constraint \( C_{\text{new}} \) is related to the input constraints, i.e.,

\[
C_{\text{new}} = f_C(C_1, \ldots, C_m),
\]

and are interested in obtaining a membership rule set for \( C_{\text{new}} \) based on the rules for the source constraints,

\[
R_{\text{new}} = f_R(R_1, \ldots, R_m).
\]

We study the requirements for \( R_{\text{new}} \) to be atomically complete w. r. t. \( C_{\text{new}} \), since if that is the case, closure \( R_{\text{new}} \) is sufficient for enforcing GAC on the constraint \( C_{\text{new}} \).

#### 5.4.1 Conjunction of Constraints

Consider two constraints \( C_1, C_2 \) on the same variables \( X \) to which are associated the rule sets \( R_1, R_2 \), resp. We are interested in the conjunctive constraint

\[
C_{\land} = C_1 \land C_2
\]

and a rule set \( R_{\land} \) associated with it. In precise notation, we examine the constraint

\[
C_{\land} = \langle C_{1R} \cap C_{2R}, X \rangle \quad \text{based on} \quad C_1 = \langle C_{1R}, X \rangle \text{ and } C_2 = \langle C_{2R}, X \rangle.
\]

The simple rule set union \( R_1 \cup R_2 \) does generally not propagate sufficiently to enforce GAC on \( C_{\land} \), even if \( R_i \) enforces GAC on \( C_i \) for both \( i = 1, 2 \). For example, consider \( (x \neq y) \land (x = y) \) constraining the variables \( x, y \in \{0,1\} \). It is
closed under any rules correct for the one of the constraints ≠ and = individually yet it is inconsistent, which GAC-enforcing rules for the conjunctive constraint (≠) ∧ (=) do detect.

Observe that any atomic rule correct for $C_\wedge$ must also be correct for one or both of $C_1, C_2$. We define

$$R_\wedge = \text{closure}(R_1 \cup R_2),$$

and employ Theorem 5.3.9.

5.4.1. **FACT.** $R_\wedge$ is atomically complete w.r.t. $C_\wedge$ if $R_i$ is atomically complete w.r.t. $C_i$ for both $i = 1, 2$. $\Box$

**Relational (1, m)-Consistency**

The generalisation to conjunctions of $m$ constraints is

$$C_\wedge = \bigwedge_{i=1}^{m} C_i \quad \text{and} \quad R_\wedge = \text{closure} \left( \bigcup_{i=1}^{m} R_i \right).$$

From the view of the set of the constituent constraints $C_i$, all on the same set of variables, the local consistency enforced is relational (1, m)-consistency [Dechter and van Beek, 1997]. Since we enforce GAC on the conjunctive constraint, an instantiation of any one variable can be extended to a solution of it, which is also a solution of each of the constituent constraints.

Enforcing GAC on the constituent constraints separately is equivalent to relational (1, 1)-consistency, a strictly weaker local consistency.

5.4.2. **EXAMPLE.** Consider the constraints $\text{and}(x, y, z)$ and $\text{or}(x, y, z)$, representing the logical operators, and their conjunction $\text{c}(x, y, z) = \text{and}(x, y, z) \land \text{or}(x, y, z)$. It has exactly the two solutions $\{0, 0, 0\}, \{1, 1, 1\}$. In the union of the rule sets for $\text{and}, \text{or}$ (such that atomic rules are subsumed as required) we find

$$z \in \{1\} \rightarrow y \neq 0 \quad \text{for and,}$$

$$x \in \{1\}, z \in \{0\} \rightarrow y \neq 0 \quad \text{for or (infeasible rule),}$$

which allow to generate the expected rule

$$x \in \{1\} \rightarrow y \neq 0 \quad \text{for and } \land \text{ or}$$

by one step of (derive). $\Box$

To generate GAC-enforcing rules for constraints that do not share all variables as required in this section, we need constraint padding.
5.4.2 Constraint Padding

In order to construct the rules for a conjunctive constraint, the participating constraints must be on the same set of variables. This can be achieved by essentially syntactically extending the individual constraints to new variables, without actually constraining them. We call such a modification padding. Extending the rules accordingly is slightly more complicated, due to newly arising infeasible rules.

So we examine \( C_p(X, v) \) such that \( C_p[X] = C \) and \( v \in D_v \). Formally

\[
C_p = \langle C_R \times D_v; \ X, v \rangle \quad \text{based on} \quad C = \langle C_R, X \rangle \quad \text{where} \quad v \not\in X \quad \text{and} \quad v \in D_v.
\]

If \( R \) is the set of rules associated with \( C \) then the rules \( R_p \) associated with \( C_p \) are found by

\[
R_p = \text{closure}(R_1 \cup R_2),
\]

where

\[
R_1 = \{ X \in S, v \in D_v \rightarrow y \neq a \mid (X \in S \rightarrow y \neq a) \in R \},
\]

\[
R_2 = \{ X \in S, y \in \{ a \} \rightarrow v \neq b \mid (X \in S \rightarrow y \neq a) \in R \land b \in D_v \}.
\]

The set \( R_1 \) pads the input rules by simply adding the redundant test \( v \in D_v \). All correct atomic rules with bodies on the variables \( X \) of \( C \) are constructed in this way. \( R_1 \) is closed under (derive), (subsume) if \( R \) is.

The set \( R_2 \) consists of rules that disallow values for the new variable \( v \). Since \( v \) is not actually constrained, no such rule can exist that is both correct and feasible, however. Therefore, all rules in \( R_2 \) are infeasible, since they are correct. Moreover, observe that every correct, atomic, infeasible rule with a body disequality on \( v \) is subsumed by \( R_2 \).

In conclusion, \( R_1 \cup R_2 \) is atomically complete w.r.t. \( C_p \) if \( R \) is atomically complete w.r.t. \( C \).

5.4.3 Fact. \( R_p \) is atomically complete w.r.t. \( C_p \) if \( R \) is atomically complete w.r.t. \( C \). \( \square \)

The pre-closure processing is linear in the size of the set \( R \).

5.4.4 Example. We pad the Boolean constraint \( \neg(x, y) \) with the extra variable \( z \in \{0, 1\} \) to \( \neg'(x, y, z) \).

\[
\neg(x, y), y \in \{0\} \rightarrow x \neq 0 \quad \neg'(x, y, z), y \in \{0\}, z \in D \rightarrow x \neq 0
\]

\[
\vdots
\]

\[
\neg'(x, y, z), x \in \{0\}, y \in \{0\} \rightarrow z \neq 0
\]

\[
\neg'(x, y, z), x \in \{0\}, y \in \{0\} \rightarrow z \neq 1
\] \( \square \)
5.4. **Cases of Incremental Rule Generation**

### Multi-Constraint Membership Rules

We are now in the position to deal with conjunctions of constraints that do not share all variables. To obtain rules for the conjunction from rules of the constraints participating in the conjunction, find the union of all their variables, extend the constraints and their rules by appropriate padding to these variables, and close the union of the resulting rule sets under the meta rules.

This insight enables us to derive multi-constraint membership rules.

#### 5.4.5. Example

Let us examine the interaction of the two logic constraints and$(x, y, z)$ and not$(x, y)$ in the conjunction $\text{and}(x, y, z) \land \text{not}(x, y)$.

Given appropriate rule sets $R_{\text{and}}, R_{\text{not}}$ for the constituent constraints, we proceed by first padding $R_{\text{not}}$ by the extra variable $z$ to $R_{\text{not}'},$ as done in Example 5.4.4. Subsequently, $R_{\text{and,not}}$ is found as closure$(R_{\text{and}} \cup R_{\text{not}})$. It contains the rule $x \in \{0,1\}, y \in \{0,1\} \rightarrow z \neq 1$ for the conjunction of and, not. More precisely, we have derived

$$\text{and}(x, y, z), \text{not}(x, y), x \in \{0,1\}, y \in \{0,1\} \rightarrow z \neq 1,$$

a multi-constraint membership rule.

In [Abdennadher and Rigotti, 2004], a propagation rule generation method is presented that is capable of producing multi-headed propagation rules directly. The method is based on a generate-and-test approach. Its purpose is the generation of rules for the interaction of constraints. For example, one can apply it to $\text{and}(x, y, z)$ and $\text{not}(u, v)$ to generate all rules with these constraints and additional equality constraints between variables from $\{x, y, z\}$ and $\{u, v\}$ in the rule condition.

We can generate equivalent (membership) rules describing all interaction patterns between constraints, by performing the corresponding rule set constructions for each pattern.

Enforcing GAC on conjunctions of constraints instead of just on the participating constraints individually can increase search efficiency, despite the additional propagation cost. That is the case especially when the participating constraints share many variables [Katsirelos and Bacchus, 2001].

#### 5.4.3 Defining a Constraint by its Non-Solutions

While constraints are often defined positively by stating their solutions, sometimes it is more natural to define a constraint negatively by stating tuples that are not solutions. Suppose that $\langle \text{Neg}, X \rangle$ is a set of tuples associated with variables $X$, and define the constraint $C_N(X)$ by

$$C_N = \langle D^n - \text{Neg}, X \rangle \quad \text{where} \quad n = |X| \quad \text{and} \quad X \in D^n.$$
A rule set that is atomically complete w. r. t. \( C_N \) can be obtained in a particularly simple way. In fact, we can precisely construct the correct atomic rules, by Note 5.3.4 which states the correspondence between an atomic rule and a non-solution. Abbreviate

\[
\text{lhs}(X, t, i) = x_1 \in \{t[x_1]\}, \ldots, x_{i-1} \in \{t[x_{i-1}]\}, x_{i+1} \in \{t[x_{i+1}]\}, \ldots, x_n \in \{t[x_n]\}
\]

and define

\[
R_{\text{Neg},i} = \{ \text{lhs}(X, t, i) \rightarrow x_i \neq t[x_i] \mid t \in \text{Neg} \}
\]

\[
R_{\text{Neg}} = \bigcup_{i \in [1..n]} R_{\text{Neg},i} \tag{5.5}
\]

The construction of \( R_{\text{Neg}} \) is linear in the size of \( \text{Neg} \): precisely \( n \cdot |\text{Neg}| \) atomic rules are produced.

We finally have

\[
R_N = \text{closure}(R_{\text{Neg}}).
\]

5.4.6. FACT. \( R_N \) is atomically complete w. r. t. \( C_N \). \( \square \)

5.4.7. EXAMPLE. Define a constraint over the variable sequence \( x, y, z \in [1..10] \) such that they do not represent an increasing sequence \( x, y, z \) of prime numbers. The 4 non-solutions are \((2, 3, 5), (2, 3, 7), (2, 5, 7), (3, 5, 7)\). The respective rules:

- 12 atomic rules:
  \[
  \begin{align*}
  x \in \{2\}, y \in \{3\} & \rightarrow z \neq 5 \\
  y \in \{3\}, z \in \{5\} & \rightarrow x \neq 2 \\
  : \\
  y \in \{5\}, z \in \{7\} & \rightarrow x \neq 3
  \end{align*}
  \]

- 8 rules after closure:
  \[
  \begin{align*}
  x \in \{2\}, y \in \{5\} & \rightarrow z \neq 7 \\
  x \in \{2\}, y \in \{3, 5\} & \rightarrow z \neq 7 \\
  : \\
  y \in \{3, 5\}, z \in \{7\} & \rightarrow x \neq 2
  \end{align*}
  \]

5.4.4 Defining a Constraint by its Solutions

When the constraint is defined positively by an explicit set of solutions (or a procedure that enumerates them), the incremental closure method can be used as well, by first converting the positive definition into a corresponding negative one. The method based on non-solutions, described in the previous section, then becomes applicable.
5.4.8. Example. Let $C$ be defined as the Boolean binary constraint $\{(0,1),(1,0)\}$. Its non-solutions are $Neg = \{(0,0),(1,1)\}$. View $C$ to be defined as $\{0,1\}^2 - Neg$ and generate the rules from the negative definition.

In this way, we have a procedure that corresponds in input and output to the RGA generation algorithm of [Apt and Monfroy, 2001]. We compare the two algorithms in detail in Section 5.6.2.

5.4.5 Enlarging the Base Domain

The following observation explains what it means to extend the variable domains by a new value. We redefine the constraint in such a way that an associated rule set needs not be modified. Assume

$$C = \langle C_R, X \rangle \text{ with } X \in D^n.$$ 

We extend the domain by the value $e$ not previously present. Let $D_e = D \cup \{e\}$, and define

$$C_e = \langle C_R \cup D^n_e - D^n, X \rangle \text{ with } X \in D^n_e.$$ 

So a tuple $t \in D^n_e$ is a solution of $C_e$ either

- if it is already a solution of $C$, or

- if it uses the new value: $t[x] = e$ for some variable $x$.

The non-solutions of $C_e$ are exactly the non-solutions of $C$. Note 5.3.4 entails the following link.

5.4.9. Fact. A given rule set is atomically complete w.r.t. $C_e$ if and only if it is atomically complete w.r.t. $C$.

Additional Modifications of the New Solution Set

A domain enlargement can be combined well with the addition of solutions or non-solutions that use the new value. The case of adding some non-solutions $Neg_e$ reduces straightforwardly to some known methods. How the non-solutions $Neg_e$ give rise to a constraint is explained in Section 5.4.3. In turn, this constraint is combined with the domain-extended constraint $C_e$ into a conjunctive constraint, following Section 5.4.1. The case of adding some new solutions $Pos_e$ can be reduced to the previous one: the solutions are turned into non-solutions by

$$Neg_e = D^n_e - D^n - Pos_e.$$ 

Note that the solutions of $C$ are irrelevant.
5.4.10. Example. We construct rules for the constraint or3, that represents disjunction in the three-valued logic 0, 1, u [Kleene, 1952]. We base the rule generation on the rules $R_\alpha$ for the conventional Boolean constraint or, and we use the fact that we can define or3 alternatively as

\[ \text{or3} = \text{or} \cup Pos_u \]

\[ Pos_u = \{(0, u, u), (u, 0, u), (1, u, 1), (u, 1, 1), (u, u, u)\}. \]

So $Pos_u$ defines the value $u$ here. Consider now the constraint $\text{or'}$ defined by

\[ \text{or'} = \text{or} \cup \{(x, y, z) \mid \{u\} \subseteq \{x, y, z\} \text{ and } x, y, z \in \{0, 1, u\}\}, \]

that is, $\text{or'}$ permits as a solution any tuple containing $u$ at some position. We have extended the or constraint by enlarging the domain by $u$. If $R_\alpha$ is atomically complete w.r.t. or then it is so w.r.t. $\text{or'}$ as well; we can write $R_\alpha' = R_\alpha$.

Next, we compute the tuples

\[ Neg_u = \{0, 1, u\}^3 - \{0, 1\}^3 - Pos_u \]

that use the value $u$ and are unacceptable as solutions of or3. We find the corresponding atomic rules in $R_{Neg_u}$ as in (5.5) of Section 5.4.3. Finally, we obtain $R_{or3} = \text{closure}(R_{or'} \cup R_{Neg_u})$. The rule set $R_{or3}$ is atomically complete w.r.t. the constraint or3.

5.4.6 Universal Quantification

Assume a constraint $C$ on some variable sequence that includes $x$. We examine the constraint that results from universally quantifying $x$.

Let

\[ C = \langle C_R; X \rangle \text{ and } X \in D^n \text{ and } X = Y, x. \]

We define a constraint on the same variables except $x$ by

\[ C_Y = \forall x. C = \langle C_{VR}; Y \rangle \text{ and } Y \in D^{n-1} \]

such that

\[ C_R = \{ t, a \mid t \in C_{VR} \text{ and } a \in D \}. \]

So whenever the $(n - 1)$-tuple $t$ is a solution of $C_Y$ then for all values $a$ in the domain of the quantified variable $x$ we have that the $n$-tuple $t, a$ is a solution to the source constraint $C$.

Surprisingly, we can obtain rules $R_Y$ for $C_Y$ from rules $R$ for $C$ by simply eliminating all references to $x$:

\[ R_Y = \text{closure}(\{ Z \in S \rightarrow y \neq a \mid (Z \in S, x \in S_x \rightarrow y \neq a) \in R \}). \]

5.4.11. Fact. $R_Y$ is atomically complete w.r.t. $C_Y$ if $R$ is atomically complete w.r.t. $C$. \qed
5.4. Cases of Incremental Rule Generation

PROOF. Consider a rule in \( R \) associated with \( C \), and the corresponding non-solution \( d \) (Note 5.3.4).

The partial solution \( d[Y] \) can clearly not be extended to a full solution by each assignment \( \{x \mapsto a\} \) with \( a \in D \); the counter example is \( a = d[x] \).

So \( d[Y] \) is a non-solution of \( C_Y \). This means that we can indeed correctly cut the rules of \( C \) down to rules for \( C_Y \).

It remains to consider completeness, that is, whether all correct, atomic rules for \( C_Y \) are subsumed. Let \( Z \in S \rightarrow y \neq a \) be such a rule. But then some rule \( Z \in S, x \in S_x \rightarrow y \neq a \) must be subsumed by \( R \). For, otherwise all \( d \) with \( \{d[Z]\} = S, d[y] = a \) were solutions, meaning that \( x \) could be all-quantified. \( \square \)

5.4.12. EXAMPLE. We take the Boolean constraint \( \text{or}(x,y,z) \), and quantify universally on \( x \). So we consider

\[
\text{or}'(y,z) = \forall x. \text{or}(x,y,z).
\]

The only solution is \((1,1)\). Indeed, both \((0,1,1)\) and \((1,1,1)\) satisfy \( \text{or} \).

Two correct rules associated with \( \text{or} \) are

\[
\begin{align*}
\text{or}(x,y,z), x \in \{0\}, z \in \{1\} & \rightarrow y \neq 0, \\
\text{or}(x,y,z), x \in \{1\}, z \in \{0\} & \rightarrow y \neq 0.
\end{align*}
\]

They lead to

\[
\begin{align*}
\text{or}'(y,z), z \in \{1\} & \rightarrow y \neq 0, \\
\text{or}'(y,z), z \in \{0\} & \rightarrow y \neq 0,
\end{align*}
\]

from which by meta rule closure

\[
\text{or}'(y,z), z \in \{0,1\} \rightarrow y \neq 0
\]

is derived. \( \square \)

The problem of enforcing GAC on all-quantified constraints is studied in [Bordeaux and Monfroy, 2002]. The authors discuss a number of Boolean constraints and associated rules for enforcing GAC, and point out the need for automatic rule generation for quantified constraints.

5.4.7 Existential Quantification

Existential quantification (projection) is the dual to the introduction of variables (padding), Section 5.4.2.

Assume a constraint \( C \) one some variable sequence that includes \( x \). We examine the constraint that results from existentially quantifying \( x \).
Chapter 5.  Incremental Rule Generation

Presume

\[ C = (C_R; X) \quad \text{and} \quad X \in D^n \quad \text{and} \quad X = Y, x. \]

We define a constraint on the same variables except \( x \) by

\[ C_3 = \exists x. C = (C_3R; Y) \quad \text{and} \quad Y \in D^{n-1} \]

such that

\[ C_3R = \{ d[Y] \mid d \in C_R \}. \]

So whenever \( d \) is a solution of \( C_3 \) then a value \( a \) in the domain of the quantified variable \( x \) exists such that the \( n \)-tuple \( t, a \) is a solution to the source constraint \( C \).

The construction of rules \( R_3 \) for \( C_3 \) is inverse to the case of all-quantification in the sense that it requires closure prior to the modification of the rules:

\[ R_3 = \{ Z \in S \rightarrow y \neq a \mid (Z \in S, x \in D \rightarrow y \neq a) \in \text{closure}(R) \}. \]

5.4.13. FACT. \( R_3 \) is atomically complete w.r.t. \( C_3 \) if \( R \) is atomically complete w.r.t. \( C \). Moreover, \( R_3 \) is closed under the meta rules (derive), (subsume).

PROOF. Consider a rule of the form \( Z \in S, x \in D \rightarrow y \neq a \) from the set \( \text{closure}(R) \). It states that it is correct to conclude \( y \neq a \) from \( Z \in S \), independent of the value of \( x \). Clearly, the rule \( Z \in S \rightarrow y \neq a \) is correct for \( C_3 \).

Conversely, consider some correct, atomic rule \( Z \in S \rightarrow y \neq a \) associated with \( C_3 \). We can conclude from it that no solution \( d \) of \( C \) exist with \( \{d[Z]\} = S \) and \( d[y] = a \). So the rule \( Z \in S, x \in D \rightarrow y \neq a \) is correct when associated with \( C \), and consequently it must be subsumed by \( \text{closure}(R) \).

Finally, notice that \( R_3 \) is closed under (derive), (subsume), since any transformation possible in \( R_3 \) would have been possible in \( R \), which is closed, using the corresponding ancestor rules.

5.4.14. EXAMPLE. We take once more the Boolean constraint \( \text{or}(x, y, z) \), and quantify now existentially on \( x \), to obtain

\[ \text{or}'(y, z) = \exists x. \text{or}(x, y, z). \]

The three solutions of \( \text{or}' \) are \{\((0,0),(0,1),(1,1)\)\}. Each can be extended to a solution of \( \text{or} \) by some \( x \) in \{0, 1\}.

The only two rules contained in the closure of all atomic rules correct for \( \text{or} \) and with \( x \in D \) in their condition are

\[ \text{or}(x, y, z), x \in D, y \in \{1\} \rightarrow z \neq 0, \]

\[ \text{or}(x, y, z), x \in D, z \in \{0\} \rightarrow y \neq 1. \]
5.5. Example: A Composed fulladder Constraint

We obtain
\[ \text{or}'(y, z), y \in \{1\} \rightarrow z \neq 0, \]
\[ \text{or}'(y, z), z \in \{0\} \rightarrow y \neq 1. \]

These two are also the only correct rules for \( \text{or}' \). Indeed, the only non-solution \((1, 0)\) of \( \text{or}'(y, z) \) corresponds to the two rules (Note 5.3.4).

No further (derive) or (subsume) is possible on the rule pair. \( \square \)

5.5 Example: A Composed fulladder Constraint

In this section we demonstrate how a rule-based GAC-enforcing solver (i.e., a set of membership rules establishing GAC) for a complex constraint can be assembled from the solvers of some base constraints. We use for this the fulladder constraint. It captures the relation linking the binary variables \( x, y, z, s, c \) in such a way that the sum of \( x, y, z \) is \( s \) with the carry bit in \( c \), i.e.,

\[ x + y + z = 2c + s \quad \text{with} \quad x, y, z, s, c \in \{0, 1\}. \]

The conventional definition using the basic constraints and, or, xor is

\[
\text{fulladder}(x, y, z, s, c) \equiv \exists c_1, c_2, s_1. \quad \text{xor}(x, y, s_1) \land \text{and}(x, y, c_1) \land \text{and}(z, s_1, c_2) \land \text{or}(c_1, c_2, c) \land \text{xor}(z, s_1, s).
\]

It can be used straightforwardly to construct a rule set for fulladder from rule sets for and, or, xor. These, in turn, can be constructed from the corresponding positive or negative definition. We sketch a possible sequence of operations, using the straightforward language in Figure 5.2 (variables are handled informally for simplicity of presentation). The input to this incremental generation are 5 rule sets: one copy of a rule set describing the or constraint on the variables \((c_1, c_2, c)\), and 2 times 2 copies describing xor, and, on \((x, y, s_1), (z, s_1, s)\) and \((x, y, c_1), (z, s_1, c_2)\), resp.

1. We begin with the first two constraints inside the conjunctive definition of fulladder. Let \( R_{xor} \) and \( R_{and} \) be the corresponding rule sets for \((x, y, s_1)\) and \((x, y, c_1)\), resp. To construct the rules for the conjunctive constraint

\[
\text{aux}_1(x, y, s_1, c_1) := \text{xor}(x, y, s_1) \land \text{and}(x, y, c_1)
\]

it is necessary to pad the constraints to

\[
\text{xor}'(x, y, s_1, c_1) \land \text{and}'(x, y, s_1, c_1)
\]
and accordingly their rules. We compute
\[
\begin{align*}
R_{\text{xor}}' & := \text{pad}(c_1, R_{\text{xor}}) \\
R_{\text{and}}' & := \text{pad}(s_1, R_{\text{and}})
\end{align*}
\]
and subsequently
\[
R_{\text{aux-1}} := \text{union}(R_{\text{xor}}', R_{\text{and}}')
\]

In other words, we obtain \( R_{\text{aux-1}} = R_{\text{xor}}' \cup R_{\text{and}}' \).

2. This pattern of padding and union-building is repeated. Let us denote the result by \( R_{\text{aux-2}} \) associated with the constraint \( \text{aux-2}(x, y, z, s, c, s_1, c_1, c_2) \).

3. It remains to eliminate the auxiliary variables \( s_1, c_1, c_2 \) by existential quantification; see Section 5.4.7. The set \( R_{\text{aux-2}} \) must be closed before the appropriate rules can be selected. We obtain
\[
R_{\text{fulladder}} := \exists\{s_1, c_1, c_2\}, \text{closure}(R_{\text{aux-2}})
\]

By this process, 94 membership rules for \text{fulladder} are constructed from 2 \cdot 12 rules for both occurrences of \text{xor} and 3 \cdot 9 rules for the two occurrences of \text{and} and the single occurrence of \text{or}. These input rules are the closure of all correct, atomic rules for their respective constraints, therefore the constructed rule set enforces GAC on \text{fulladder}. This in turn means strictly more propagation than GAC on the 5 individual constraints. The CSP \(<\text{fulladder}(x, y, z, s, c); x, z, c \in \{0, 1\}, y \in \{1\}, s \in \{0\}>\) is closed under the subconstraint rules - there is no constraint and thus no rule linking \( y, s \) directly. In contrast, one rule constructed for \text{fulladder} is \( y \in \{1\}, s \in \{0\} \rightarrow c \neq 0 \) which allows to propagate to \( c \in \{1\} \).

## 5.6 Implementing the Meta Rule Closure

### 5.6.1 Uniqueness

To show that the meta rule application strategy has no influence on the result of the closure, we view \((\text{derive}), (\text{subsume})\) as a rewrite system. For the relevant background we refer to [Baader and Nipkow, 1998].

5.6.1. **Lemma.** The closure of a membership rule set under \((\text{derive}), (\text{subsume})\) exists and is unique.
5.6. Implementing the Meta Rule Closure

PROOF. If the number of variables and the size of their base domains is finite, then there are only finitely many syntactically correct rules. Any closure algorithm that applies \((\text{derive})\), \((\text{subsume})\) at most once to any pair of rules and a specific \(k\) in the input rule set \(R\) terminates.

The meta rule system is confluent. We prove this by showing that every critical pair is joinable. The rule sets \(R_1, R_2\) in a critical pair \(\langle R_1, R_2\rangle\) are the respective results of applying two meta rules to the same source rule set.

Joinability of critical pairs is easy to verify for pairs stemming from \((\text{subsume})+(\text{subsume})\) and \((\text{derive})+(\text{derive})\). For the meta rules \((\text{subsume})+(\text{derive})\), the interesting case is the critical pair arising from the source rule set \(R \cup \{r_a, r_b, r_c\}\) and where \(r_a\) subsumes \(r_b\), and \(r_b, r_c\) have a descendant rule \(r_d\). An application of the subsumption meta rule can here prevent a subsequent application of the derivation meta rule. We find the two possible initial derivations

\[
\frac{R \cup \{r_a, r_b, r_c\}}{R \cup \{r_a, r_c\}} \quad \text{(subsume)} \quad \text{on } r_a, r_b
\]

\[
\frac{R \cup \{r_a, r_b, r_c\}}{R \cup \{r_a, r_c, r_d\}} \quad \text{(derive)} \quad \text{on } r_b, r_c
\]

so the critical pair is \(\langle R \cup \{r_a, r_c\}, R \cup \{r_a, r_b, r_c, r_d\}\rangle\). We show this pair to be joinable by a case distinction on whether \(r_a\) subsumes \(r_d\).

\(r_a\) subsumes \(r_d\). The two derivations (continuing the ones above)

\[
\frac{R \cup \{r_a, r_c\}}{R \cup \{r_a, r_c\}} \quad \text{(subsume)} \quad \text{on } r_a, r_b
\]

\[
\frac{R \cup \{r_a, r_b, r_c, r_d\}}{R \cup \{r_a, r_c, r_d\}} \quad \text{(subsume)} \quad \text{on } r_a, r_d
\]

show that \(R \cup \{r_a, r_c\}\) and \(R \cup \{r_a, r_b, r_c, r_d\}\) are joinable.

\(r_a\) does not subsume \(r_d\). In this case, from \(r_a\) and \(r_c\) a descendant \(r_e\) can be derived. In turn, \(r_e\) subsumes \(r_d\). For the critical pair, we have the following two continuations of the above derivations:

\[
\frac{R \cup \{r_a, r_c\}}{R \cup \{r_a, r_c, r_e\}} \quad \text{(derive)} \quad \text{on } r_a, r_c
\]

\[
\frac{R \cup \{r_a, r_b, r_c, r_d\}}{R \cup \{r_a, r_c, r_d\}} \quad \text{(subsume)} \quad \text{on } r_a, r_b
\]

\[
\frac{R \cup \{r_a, r_b, r_c, r_d\}}{R \cup \{r_a, r_c, r_d\}} \quad \text{(subsume)} \quad \text{on } r_a, r_c
\]

So \(R \cup \{r_a, r_c\}\) and \(R \cup \{r_a, r_b, r_c, r_d\}\) are joinable in \(R \cup \{r_a, r_c, r_e\}\).

Since the rewrite system \((\text{derive}), (\text{subsume})\) is terminating and confluent, the closure of a rule set under these meta rules exists and is unique. \(\square\)

Lemma 5.6.1 allows us to apply the \((\text{subsume})\) meta rule eagerly when computing the closure, which improves convergence.
Chapter 5. Incremental Rule Generation

[RGA]: constraint $C \subseteq D^n$ on $X \rightarrow$ rule set enforcing GAC on $C$

$$R := \emptyset$$

for each $V \subseteq X$ in increasing order do
  for each $S \subseteq D^{|V|}$ in decreasing order, where $S_i \subseteq C[y_i]$ and $S \cap C[V] \neq \emptyset$, do
    for each $y \in X - V$ and each $d \in D$ do
      let $r$ be the rule $C(X), V \in S \rightarrow y \neq d$
      if $r$ is correct and not subsumed by $R$ then
        $R := R \cup \{r\}$
    end
  end
end

return $R$

Figure 5.1: Original rule generation algorithm RGA [Apt and Monfroy, 2001]

5.6.2 Relation to the Original Generation Algorithm for Membership Rules

We have already seen in Section 5.3.4 that RGA sometimes does not find every ‘interesting’ rule. We inspect here in detail its relation to the closure-based approach to rule generation, which can be used in the same way as RGA to generate rules, by the method described in Section 5.4.4.

RGA implements essentially a generate-and-test approach, where the rule candidates are ordered by subsumption such that the output rule set grows steadily as unsubsumed rules are added. We quote RGA in Figure 5.1.

5.6.2. LEMMA. For a given constraint $C$, denote by $R_{RGA}$ the rule set that RGA generates. Let $R_{cl}$ be a set of correct rules associated with $C$ such that every atomic rule correct for $C$ is subsumed, and $R_{cl}$ is closed under (derive), (subsume). Then

- every rule in $R_{RGA}$ is subsumed by some rule in $R_{cl}$, and
- every feasible rule in $R_{cl}$ subsumes some rule in $R_{RGA}$.

PROOF. RGA enumerates all correct, feasible rules, discarding those that are subsumed. In turn, $R_{cl}$ subsumes all correct rules, by Theorem 5.3.9.

Inversely, each feasible rule in $R_{cl}$ is correct, and not subsumed by a different correct rule. This means that it is either contained in $R_{RGA}$, or subsumes a rule therein. The latter case arises due to infeasible rules and (derive). \qed
5.7 Implementation and Empirical Evaluation

We implemented a prototype of incremental rule generation in the ECL\'PS* system. The program accepts rule generation requests in the language described in Fig. 5.2 (where, for simplicity, domain and variable handling is omitted). The rule set closure is computed by the algorithm shown in Fig. 5.3. We argue for its correctness briefly and informally by stating that $R$ in the algorithm remains always closed under (derive), (subsume), and that any possible (derive) between two rules of $R$ is collected (stage-wise) in $D$. Since upon termination $D$ has been emptied into $R$ by addition or subsumption, we then have that $R$ is closed under (derive), (subsume) and that all rules of the input rule set are subsumed by $R$.

**Benchmarks**

We examined the behaviour of the closure algorithm for the generation of rule sets from positive, extensional constraint definitions. This enables a direct comparison with the RGA algorithm of [Apt and Monfroy, 2001].

We used random constraints with uniformly distributed solutions. We varied the tightness of the constraint, i.e. the proportion of non-solutions (a small tightness means many solutions), and we examined varying arity and domain size. Our random constraint generator is based on the program [Bessière, 1996] which was adapted so as to generate a single, $n$-ary constraint definition. Per data point we used 5 random constraints and 3 repetitions for each (we found only small variances in the measured times).

The results, summarily reported in Fig. 5.4, indicate that the closure-based rule generation approach is more efficient than RGA by orders of magnitudes when the constraint arity or the tightness is small.

---

```
rule-set ::= positive_definition((tuple-set))
| negative_definition((tuple-set))
| pad((new-variable), (rule-set))
| enlarge_domain((new-value), (rule-set))
| exists((variable), (rule-set))
| for_all((variable), (rule-set))
| union((rule-set), (rule-set))
| closure((rule-set))
```

Figure 5.2: A language for incremental membership rule generation
Chapter 5. Incremental Rule Generation

\[ \text{closure set } R \rightarrow \text{closure}(R) \]
\[ \text{return } \bigcup_{v_y,v_a} \text{closure_split}\{ \{ r \in R \mid r = (h \rightarrow y \neq a) \text{ for some } h \} \} \]

\[ \text{closure_split : rule set } R \rightarrow \text{closure}(R) \]
\[ \text{assumes that all rules have the same body} \]
\[ \text{if } R = \emptyset \text{ then return } \emptyset \]
\[ \text{else} \]
\[ \text{choose } r \in R \]
\[ \text{return } \text{closure_add}(R - \{r\}, \{r\}) \]
\[ \text{end} \]

\[ \text{closure_add : rule sets } A, R \rightarrow \text{closure}(A \cup R) \]
\[ \text{assumes that } R \text{ is closed} \]
\[ \text{if } A = \emptyset \text{ then return } R \]
\[ \text{else} \]
\[ D := \emptyset \]
\[ \text{for each } r \in A \text{ not subsumed by } R \text{ do} \]
\[ \langle D_r, R \rangle := \text{closure_add_one}(r, R) \]
\[ D := D \cup D_r \]
\[ \text{return } \text{closure_add}(D, R) \]
\[ \text{end} \]

\[ \text{closure_add_one : rule } r, \text{ rule set } R \rightarrow \text{rule set pair } \langle D, R' \rangle \]
\[ D \text{ is the set of descendants between } r \text{ and } R, \text{ and} \]
\[ R' \text{ is } R \text{ updated with } r; \text{ always considering subsumption} \]
delete all rules subsumed by \( r \)
\[ D := \text{all (derive) descendants of } r \text{ and any rule in } R \]
\[ \text{if some } r' \in D \text{ subsumes } r \text{ then} \]
\[ \text{return } \text{closure_add_one}(r', R) \]
\[ \text{else} \]
delete all rules subsumed by \( D \)
\[ \text{return } \langle D, R \cup \{r\} \rangle \]
\[ \text{end} \]

Figure 5.3: Algorithm to close a membership rule set under (derive), (subsume)
5.8 Final Remarks

We presented an incremental approach to the automatic generation of constraint propagation rules; and we examined it in depth for the case of membership rules. The closure of a rule set under two meta rules constitutes the core of this method. We showed that various ways of defining a constraint incrementally have a corresponding rule generation method. We could also demonstrate the efficiency of this method. That, and the description by meta rules, may make it possible to include it as a source-to-source transformation process during the compilation of CHR rules [Holzbaur, 2002].

An important contribution of this work is that it helps to further explain rule-based constraint propagation. For the class of membership rules, the close connection between an atomic unit of the constraint definition, i.e. a single non-solution, and a GAC-enforcing rule application is visible. Evidence for the importance of the explanatory aspect of the rule-based view on constraint propagation is the recent work [Choi et al., 2003], in which propagation rules are employed as a means to argue for operational relevance or redundancy of constraints in dual-model CSPs.

While a complete meta rule closure of a rule set subsuming all correct, atomic rules is necessary to obtain a GAC-enforcing propagation operator for a constraint, compromising on these conditions (except for correctness) does not impede correctness of the corresponding propagation. Situations are conceivable in which it is useful to enter only some of the atomic rules into the closure, or in which the closure is executed incompletely. The resulting solver generally propagates to a local consistency weaker than GAC, but it may consist of less, simpler rules and may therefore be faster to execute – a common trade-off in constraint programming.

Finally we note that dynamic updating of solvers may be of interest in “open-world” constraint satisfaction [Faltings and Macho-Gonzalez, 2002], where gathering the constituents of a CSP, e.g. the tuples defining a constraint, is part of the problem solving process. This is the case, for example, when a CSP is represented in a distributed way on several internet sites.

Figure 5.4: Rule generation from positive random constraint definitions