Abductive and Constraint Logic Programming

Gerhard Wetzel

May 1997

Submitted to the University of London as a thesis for the degree of Doctor of Philosophy

Department of Computing
Imperial College of Science, Technology and Medicine
Abstract

This thesis proposes a novel framework called CALOG: "C" for constraints, "A" for abduction, "LOG" for logic. The framework was motivated by the aim to unify the areas of standard Logic Programming (LP, [Ho90, Ko79b, Li87]), Abductive Logic Programming (ALP, [CoThTo91, KaKoTo93, To95]), Constraint Logic Programming (CLP, [JaLa87, VH91, JaMa94]), and Semantic Query Optimization (SQO, [ChGrMi87, ChGrMi90]).

The CALOG framework achieves this aim by combining the use of definitions, as in standard LP, with the use of additional integrity constraints, as in ALP and SQO. While definitions are executed in conventional logic programming goal reduction manner, integrity constraints are executed in forward reasoning style to check potential answers for consistency, similarly to Constraint Handling Rules (CHR, [Fr92, Fr95]) in CLP.

It is hoped and will be illustrated by several examples that the extension of standard LP by integrity constraints allows a more direct and hence more natural translation into logic of many procedurally specified algorithms. Another important advantage of the unified framework is its applicability to a wide range of problems, possibly including some inter-disciplinary applications for which neither of the framework instances alone would be particularly suitable.

Potential applications of the CALOG framework include constraint satisfaction problems (such as the n-queens problem or map colouring), Operations Research applications (such as job-shop scheduling and warehouse location problems), and general AI and expert system applications (terminological reasoning, configuration problems, planning and other problems represented in...
the situation/event calculus etc.).

The CALOG framework and its proof procedure should also be viewed as laying the foundation of a new programming language, tentatively called *PROCALOG* (*PROgramming with Constraints and Abducibles in LOGic* or simply *PROCrogramming in CALOG*). Two preliminary implementations of PROCALOG, one in PROLOG and one in an extension of PASCAL, have been produced as a result of this research and have been used to study in practice several of the aforementioned example applications. First computational results are reported.
Acknowledgements

I am very grateful to Robert Kowalski for supervising this work. A lot of his ideas, communicated in numerous long meetings, have gone into this thesis. In spite of his many other commitments he found time to give me detailed and knowledgeable comments on several drafts of the thesis, helping me identify several weak or wrong arguments.

Francesca Toni deserves credit for actively participating in parts of this research. She also invested a substantial amount of time in providing me with sometimes highly critical and unrelenting feedback for which I thank her very much.

The Department of Computing at Imperial College and particularly the Logic Programming Section formed a fruitful setting for this research. I am grateful for many useful discussions about my own and related research with many colleagues in the department and with visitors. Particularly helpful were the discussions with my office mates Eric Fung (during the initial phase of my work), Jacinto Dávila and Yongyuth Permpoontanalarp.

I would also like to thank Ken McAloon for a number of helpful remarks during the final phase of finishing the thesis at the Logic Based Systems Lab at Brooklyn College and Marc Denecker for making the latest version of his work on SLDNFA available to me and for explaining some of its features.
Contents

1 Introduction
1.1 General introduction ........................................ 11
1.2 Aims of the thesis .......................................... 15
1.3 Overview of the thesis ....................................... 17

2 Unifying Framework and Semantics .......................... 29
2.1 Knowledge representation .................................... 29
  2.1.1 Definitions and integrity constraints .................. 29
  2.1.2 Predicates ............................................ 34
2.2 Computational task .......................................... 36
  2.2.1 Goals ................................................. 36
  2.2.2 Suspension ............................................ 38
  2.2.3 Framework instances ................................. 43
2.3 Declarative semantics ....................................... 45
  2.3.1 Intended models ...................................... 45
  2.3.2 Answer definition(s) ................................. 46
2.4 Integrity constraint satisfaction .......................... 51
  2.4.1 The propertyhood view ............................... 51
  2.4.2 Comparison with other views of IC satisfaction .... 52
  2.4.3 Integrity constraints for equality .................... 55
2.5 Semantics for normal LP ................................. 58
3 Proof Procedure 63

3.1 The CALOG Proof Procedure (CPP) 64
  3.1.1 Operations of the proof procedure 64
  3.1.2 Extracting computed answers 70
  3.1.3 Safety of operations 76

3.2 Order of the operations 81

3.3 Soundness results 82
  3.3.1 Soundness of the operations 83
  3.3.2 Affirmation soundness 84
  3.3.3 Refutation soundness 91

3.4 Completeness 92
  3.4.1 Incompleteness of the general CPP 92
  3.4.2 The negation-free LP case 94
  3.4.3 The general case 96

4 Unified Areas 99

4.1 Semantic Query Optimization 99
  4.1.1 Deductive databases and SQO semantics 100
  4.1.2 Procedures for SQO 101
    4.1.2.1 The compilation phase 102
    4.1.2.2 The transformation phase 104

4.2 Abductive Logic Programming 107
  4.2.1 Declarative semantics 107
  4.2.2 Proof procedures 110
    4.2.2.1 Underlying frameworks 111
    4.2.2.2 Semantics 112
    4.2.2.3 Operations corresponding to CPP operations 115
    4.2.2.4 Extraction of computed answers 117
    4.2.2.5 Generalized splitting 120
    4.2.2.6 Factoring 122
6.4.2 Knowledge representation in the CPS approach .......... 212
6.4.3 Executing a goal ............................................. 216

6.5 Logic programming with integrity constraints .......... 218
6.5.1 Local propagation for Boolean operators .......... 218
6.5.2 Computing Fibonacci numbers bottom-up .......... 219

6.6 Other applications ............................................ 221
6.6.1 Legal reasoning ............................................. 221
6.6.2 The event calculus ........................................... 222
6.6.3 Reconciling rational and reactive agents .......... 223

7 Conclusion ......................................................... 225
Chapter 1

Introduction

This chapter first provides a general introduction to the areas of (standard) Logic Programming, Abductive Logic Programming, Constraint Logic Programming and Semantic Query Optimization. In pointing out close links between these areas, the idea of a unifying framework is motivated. This is one of the four major aims of the thesis which are presented in section 1.2. Finally, section 1.3 contains a detailed overview of the remaining chapters of the thesis.

1.1 General introduction

Logic Programming (LP, [Ho90, Ko79b, Li87]) is based both on the declarative representation of knowledge in terms of Horn clauses

\[ H \leftarrow B_1 \land \ldots \land B_n \]

(where \( H \) and the \( B_i \) are atoms) and on the procedural interpretation of such clauses as goal-reduction procedures

"To prove the conclusion \( H \), prove the conditions \( B_1 \) and \( \ldots \) and \( B_n \)."

A set of Horn clauses has a unique minimal model and there is a sound and complete proof procedure (with respect to the minimal model semantics), namely SLD resolution, which reduces goals by backward reasoning in accordance with this procedural interpretation.

The original definition of LP has been extended to normal LP (or standard LP) by allowing negative literals as conditions of clauses. Regarding negation
as failure and using the Clark completion [CI78] of a normal logic program is one possibility of giving a semantics to normal LP. Several other semantics have meanwhile been proposed (see the surveys [ApBo94, Sh88] and the discussion in section 2.5).

Many extensions of normal LP have been proposed. [LiTo84] showed that an extended clause

\[ H \leftarrow F \]

where \( F \) is an arbitrary sentence of first-order logic, can be transformed into a normal logic program, but at the same time argued that programs using extended clauses are preferable because they "allow the solution of many problems to be expressed in a form similar to the specification of the problem". This argument will also be made in this thesis, but the extension of normal LP to be proposed here is much more conservative than the extended clauses of [LiTo84]: it is the use of integrity constraints

\[ A_1 \land \ldots \land A_n \rightarrow B_1 \lor \ldots \lor B_m \]

(where the \( A_i \) and \( B_j \) are atoms) for integrity checking by forward reasoning, i.e.

"If the conditions \( A_1 \) to \( A_n \) hold, then derive the conclusion \( B_1 \lor \ldots \lor B_m \)."

While it is possible to transform integrity constraints into normal logic programs [KoSa88, KoSa90], operationally useful information may be lost in the transformation and the resulting program may not only be further from the specification (and thus harder to understand and to prove correct), but also less efficient to run. Moreover, integrity constraints are similar to general if-then or condition-action rules which have proven to be a popular tool in Artificial Intelligence (see, e.g., [Wi92] and [RuNo95]). Large-scale commercial AI applications such as R1/XCON [MD82] make use of such rules. The availability of integrity constraints in an LP framework might thus make the framework more attractive for AI application developers.

Another important application of integrity constraints is to check the consistency of deductive databases which are themselves an important application
of LP. The predicates of a deductive database are often divided into *extensional* and *intensional* ones, and the database is accordingly separated into an extensional database (EDB) of facts (ground Horn clauses without conditions) and an intensional database (IDB) which corresponds to a normal logic program. Given a query to a deductive database it may be more efficient to avoid accessing the EDB because looking up information in the EDB may be computationally explosive if the EDB is very large. Therefore it is desirable to transform — or optimize — the query using only the IDB and integrity constraints. It may even be possible to show that a query has no answers without accessing the EDB. The area of research concerned with such query transformations has been called Semantic Query Optimization (SQO, [ChGrMi88, ChGrMi90]).

Example 1.1.1 (SQO)
Query: `employee(X) ∧ position(X,manager) ∧ bonus(X,B) ∧ B=0`
Integrity constraint: `position(X,manager) ∧ bonus(X,B) → B≠0`
Suppose that `employee`, `position` and `salary` are all defined in the EDB. SQO may use the integrity constraint to show that the query has no answers without accessing the EDB. If PROLOG were used to process the query, it would have to look at every individual `employee` record.

The optimization of a query by means of integrity constraints may introduce new atoms of EDB predicates into the query. [Ka91a, Ka91b] pointed out that it is possible to establish a correspondence between EDB predicates and the abducibles in Abductive Logic Programming (ALP, [KaKoTo93, CoThTo91, KaMa90, To95]).

The conventional abductive framework is a triple `(T, Ab, IC)` with `T` being a logic program, `Ab` a set of predicate symbols (called *abducibles*) and `IC` a set of implications (called integrity constraints). In deductive database terminology, `T` represents the IDB, the abducibles correspond to the EDB predicates, and `IC` corresponds to the database integrity constraints.
Abduction is hypothetical, explanatory reasoning: given an observation or goal $G$, find an abductive hypothesis $\Delta$ explaining $G$. This can be formalized as follows:

1. $\Delta$ consists of atoms of abducibles only
2. $T \cup \Delta \models G$
3. $T \cup \Delta$ satisfies the integrity constraints $IC$

Different ways of defining integrity constraint satisfaction will be discussed later (see esp. section 2.4). At the very least it means that $T \cup \Delta$ has to be consistent with $IC$ (consistency view).

Example 1.1.2 (ALP)

$T: \text{bird} \leftarrow \text{albatross}$

$\text{bird} \leftarrow \text{penguin}$

$Ab: \{ \text{penguin, albatross, flies} \}$

$IC: \text{penguin} \wedge \text{flies} \rightarrow \text{false}$

Let the goal $G$ be

$\text{bird} \wedge \text{flies}$.

Then the explanation

$\Delta_1 = \{ \text{albatross, flies} \}$

satisfies (1)-(3) under the consistency view whereas the explanation $\Delta_2 = \{ \text{penguin, flies} \}$ is inconsistent with the integrity constraint and violates condition (3) under any view of integrity constraint satisfaction.

As integrity constraints are used for consistency checking in ALP (as well as in SQO), they are also useful for handling constraints in Constraint Logic Programming (CLP, [JaMa94, VH91, JaLa87]). Just like EDB predicates in SQO and abducibles in ALP, constraint predicates are not processed by backward reasoning with their definitions. Instead, atoms of constraint predicates are checked for satisfiability, either by a built-in constraint solver or by explicit
integrity constraints in the form of user-definable Constraint Handling Rules (CHRs, [Fr95, Fr92]).

Example 1.1.3 (CLP)
Goal: \( X > 1 \land 1 > Y \land Y > X \)
\( IC : X > Y \land Y > Z \implies X > Z \)
\( X > N \land N > X \implies false \)

A CLP system working over a finite integer domain (such as cc(FD) [VH-SaDe93]) or the domain of real numbers (CLP(\( \mathbb{R} \)), [JaLa87]) does not need the explicit integrity constraints as its constraint solver recognizes that the constraints in the goal are unsatisfiable over their respective domains.

Used as CHRs, the first integrity constraint (transitivity) adds \( X > Y \land 1 > X \) to the goal. The second integrity constraint then becomes applicable and generates \text{false}.

Note that standard LP, which cannot make use of the integrity constraints, would either refuse to process the uninstantiated arithmetic expressions or fail to terminate if it tried to unfold a separately given definition of "\( > \)".

1.2 Aims of the thesis

The proposed framework is related to several other frameworks: in ALP especially to Fung’s Iff Proof Procedure [Fu95, Fu96, FuKo96] and Denecker and DeSchreye’s SLDNFA [DeDS92, DeDS97]; in CLP especially to Frühwirth’s CLP+CH framework (CLP with explicit constraint handling by CHRs) [Fr94, Fr95] and to Smolka’s framework [Sm91] for guarded rules. The thesis continues previous work [We94, Fu93, Th92], and some of the results presented in this thesis have already been published in joint articles [KoToWe94, WeKoTo95, WeKoTo96].

The thesis has been motivated by four major objectives, which are either only partially realized or not addressed in the related frameworks mentioned above, and which will be shown in this thesis to be closely linked:
1. to unify standard LP, ALP, CLP, and SQO;

2. to incorporate global problem-solving strategies into LP languages;

3. to extend standard LP in such a way that procedural knowledge can be represented more naturally and executed more efficiently

4. to lay the basis for a new programming language and implement a prototype to test it on some sample applications

The first aim involves the introduction of several new concepts, among others the concept of suspension (as known from concurrent LP) and a new view of integrity constraint satisfaction. There have already been other proposals to relate ALP to deductive databases (and thus potentially to SQO) [Ka91a, Ka91b] and to unify ALP and CLP [KaMi95, Mm92], but they were mostly concerned with particular aspects of the unification or developed hybrid systems rather than a new unifying framework.

The second aim, which was originally formulated in [Ko79b], is of both theoretical and practical interest. The proposals of [Ko79b] included adding integrity constraints to logic programs and using them both for adding surrogate subgoals and for deleting redundant subgoals. Additional problem-solving strategies have been proposed in [Ko92b]. They require the explicit representation of disjunctions and reasoning within and across disjunct boundaries. Methods similar to several techniques for handling "disjunctive constraints" in CLP (see [JoSo93] for an overview of such techniques) will be proposed to enable such reasoning.

The third and fourth aims are of practical significance. Their realization requires identifying and studying examples and applications and eventually an implementation of a new language. Hopefully, a more direct declarative representation of procedural knowledge also helps to avoid the necessity, often felt by PROLOG programmers and by developers of hybrid (declarative and procedural) programming languages, of mixing logic and control in order to
achieve efficiency. The aim of having a clear separation of logic and control [Ko79a] may thus be easier to achieve in the new framework.

The thesis will argue that the four aims can be achieved together in a unifying framework, employing a theorem-proving approach which allows the use of a subset of first-order logic bigger than standard LP — the major extension being integrity constraints. From a theorem-proving point of view the framework is thus also related to the SATCHMO theorem-prover [MaBr88] whose forward reasoning clauses are integrity constraints. Traditional problems from LP, ALP, CLP, and SQO as well as interdisciplinary AI problems thus constitute obvious choices for applications, and several case studies will be conducted.

1.3 Overview of the thesis

Following is an outline of the chapters of the thesis. The presentation is kept on a descriptive, informal level so that the reader may become familiar with the most important concepts and ideas without being distracted by the technical details of precise definitions (which are given in the actual chapters).

The major contribution of the thesis is chapter 2, the definition of the unifying framework and its declarative semantics. Chapter 3, the proof procedure, generalized and simplifies previous work (described in chapter 4). Chapter 5 defines the disjunctive propagation techniques as a further important contribution of the thesis. Chapter 6 presents applications. It develops a methodology for using the framework by applying it to several well-known problems from different areas of Artificial Intelligence and Logic Programming. Two existing prototype implementations demonstrate the practicability of the approach and are used to provide some computational results.

Readers who feel they are already familiar with the areas of research covered in this thesis may choose to skip the overview of the individual chapters and proceed directly to chapter 2 — reading of the remainder of this introductory
chapter will not be assumed in the rest of the thesis.

Chapter 2: Unifying Framework and Semantics

The unifying framework to be proposed has been called CALOG — C for constraints, A for abducibles and LOG for logic.

Knowledge in the CALOG framework is represented by if-and-only-if definitions of the form

$$H \leftrightarrow D_1 \lor \ldots \lor D_n, \ n \geq 1$$

where $H$ is an atom and each $D_i$ is a conjunction of literals, and by integrity constraints of the form

$$\text{Cond} \rightarrow \text{Conc},$$

where $\text{Cond}$ is a conjunction of atoms and $\text{Conc}$ is a disjunction of atoms.

Definitions and integrity constraints may involve atoms of three kinds of predicates:

- **user-defined** predicates, defined by a theory $T_u$ of user-provided definitions,
- **built-in** predicates, implicitly defined by a theory $T_b$ of built-in definitions,
- **external** predicates, defined by an inaccessible or unknown theory $T_e$ of definitions.

The union $T_u \cup T_b \cup T_e$ of all three theories is denoted by $T$.

User-defined predicates correspond to ordinary predicates in standard LP, intensional predicates in SQO, defined predicates (or non-abducibles) in ALP, and non-constraint predicates in CLP.

Built-in predicates, which include equality as well as arithmetic inequality and arithmetic operators, correspond to constraint predicates in CLP and to equality in standard LP and ALP (equality is the single constraint predicate in LP from the point of view of the CLP($X$) scheme).
External predicates correspond to abducibles in ALP and extensional predicates in SQO. As their definitions are inaccessible, knowledge about external predicates can only be provided by integrity constraints.

Integrity constraints are also used to express properties of the accessible definitions. Any sentence true in all intended models of a theory $T$ is called a property of $T$. Making properties of definitions explicit in the form of integrity constraints has several practical advantages:

- it can help speed up program execution by providing reasoning shortcuts;
- it allows forward reasoning rules to be expressed without translating them into (backward reasoning) definitions, and thus
- it provides a new way of formulating algorithms in logical form.

The initial goal is a conjunction of literals with all variables being free. The computational task, whose formalization into a proof procedure is presented in chapter 3, is to reduce a given initial goal to a disjunction of answers, which is equivalent to the initial goal in the sense that the goal and the disjunction are satisfied, in $T$, by the same assignments to the free variables. Such answers are obtained by using definitions to unfold atomic goals to equivalent disjunctions of goals which can then be split using distributivity. Surrogate subgoals are introduced by propagation (resolution) with integrity constraints. As in Operations Research, surrogate subgoals are logically redundant, but may be easier to solve.

Atoms cannot be unfolded further if they are suspended. An atom $A$ is suspended if its definition is inaccessible or if there is no accessible definition $H \leftarrow D_1 \lor \ldots \lor D_n$ such that $A$ is an instance of $H$ (i.e. $A = H\sigma$ for some substitution $\sigma$). Otherwise $A$ is called reducible.

Thus atoms of external predicates are always suspended, whereas atoms of user-defined and built-in predicates are suspended if they are insufficiently instantiated to be an instance of the head of a definition. Suspension can be
used to control the search space and the user should write definitions with search considerations in mind.

Example 1.3.1 Let the user-defined predicate \( p \) be defined by the following two definitions:

\[
\begin{align*}
p(X, a) &\iff q(X) \\
p(X, b) &\iff r(X)
\end{align*}
\]

Then \( p(X, Y) \) and \( p(a, Y) \) are suspended, but \( p(X, a) \) and \( p(a, b) \) are reducible.

If the definition was instead

\[
p(X, Y) \iff (Y=a \land q(X)) \lor (Y=b \land r(X))
\]

then atoms of \( p \) would never be suspended.

Built-in definitions are assumed to be such that the atoms \( X<Y \), \( X>0 \) and \( \text{plus}(2, X, Y) \) are all suspended, but \( 1<0 \), \( 1>0 \) and \( \text{plus}(2, 2, X) \) are not suspended (and can be reduced to \text{false}, \text{true} and \( X=4 \), respectively).

Integrity constraints are used to process suspended goals in order to decide, if possible, whether a conjunction of suspended goals is satisfiable. Propagation with integrity constraints may add reducible atoms to the goal which may allow the computation using definitions to proceed.

Answers to goals have to satisfy the integrity constraints, and an important feature of the CALOG framework is the propertyhood view of integrity constraint satisfaction. It requires the integrity constraints to be properties of the definitions, i.e. to be true in the intended models of the theory \( T \), notation \( T \models_{\text{int}} IC \).

The propertyhood view thus depends on the chosen notion of intended model. If \( T \) corresponds to a Horn clause theory, the simplest choice is the unique minimal Herbrand model. Then

\[
p \rightarrow \text{false}
\]
is a property (but not a theorem) of the theory containing only

\[
p \iff p
\]

In the general case, when \( T \) corresponds to a normal logic program containing
negative literals, different notions of intended model have to be considered, see section 2.5.

The framework's declarative semantics introduces three different notions of answer so that the proof procedure will be sound with respect to at least the weakest notion. The strongest notion is that of ideal answer. A is an ideal answer to the initial goal $G_0$ if

(A1) $A$ is a conjunction of suspended atoms and implications (whose atoms are also suspended)

(A2) $\mathcal{T} \models \forall \text{init } \forall[G_0 \leftrightarrow A \lor \text{Rest}]$ for some formula $\text{Rest}$

(A3) $\mathcal{T} \not\models \forall \text{init } \exists \neg A$

Here $\forall$ and $\exists$ denote the universal and existential closures, respectively, quantifying over all free variables.

Condition (A1) means that goals have to be reduced as far as possible, i.e. until all subgoals are suspended.

Condition (A2) says that $G_0$ and $A \lor \text{Rest}$ have to be true in the intended models of $\mathcal{T}$ for the same values of the free variables in $G_0$.

Condition (A3) says that $A$ is satisfiable in at least one intended model of $\mathcal{T}$. This entails that $\mathcal{I}C \cup \exists A$ is consistent because by assumption $\mathcal{T} \models \forall \text{init } \mathcal{I}C$.

The given answer definition is idealistic in two ways. Firstly, $\mathcal{T}$ contains the inaccessible part $\mathcal{T}_e$ which is not available (to the proof procedure) for checking conditions (A2) and (A3). Secondly, condition (A3) cannot generally be verified in practice because suspended atoms can be tested for consistency only by integrity constraints, and the integrity constraints may not provide a complete axiomatization of the theory (or, in CLP terminology, they may not be satisfaction complete).

Thus, in addition to ideal answers, two weaker answer notions will be defined in the thesis (see definition 2.3.2).
Chapter 3: Proof Procedure

This chapter describes a rather abstract (high-level) proof procedure for the CALOG framework which can be regarded as a generalization and simplification of the IfP Proof Procedure [Fu96] and SLDNFA [DeDS97]. By mainly ignoring efficiency issues, the proposed proof procedure may serve as a scheme for a family of proof procedures which can be obtained as instances by restricting and extending the abstract procedure in different ways (some of these possibilities are discussed in chapter 5). Soundness and (partial) completeness results with respect to the three answer notions from chapter 2 are presented.

A derivation of a goal $G_n$ from an initial goal $G_0$ is a finite sequence of goals $G_0, G_1, \ldots, G_n$ where $G_1$ is obtained from $G_0$ by conjoining all integrity constraints in $IC$ to $G_0$, and, for $0 < i < n$, $G_{i+1}$ is obtained from $G_i$ by applying one of the following operations:

- **unfolding**
- **propagation**
- **logical equivalence transformation**
- **equality rewriting**

Unfolding replaces a reducible atom $A$, which is either a conjunct of the goal or is a condition of an implication in the goal, by its definition. I.e. if there is a substitution $\sigma$ such that $A = H\sigma$ and $H \leftarrow D_1 \lor \ldots \lor D_m$ is an accessible definition in $T$, then replace $A$ by $(D_1 \lor \ldots \lor D_m)\sigma$.

Propagation is a form of resolution. If a subgoal contains the conjuncts $p(s_1, \ldots, s_n) \land Cond \rightarrow Conc$ and $Susp \rightarrow p(t_1, \ldots, t_n)$, then the “resolvent” $Cond \land Susp \land (s_1 = t_1 \land \ldots \land s_n = t_n) \rightarrow Conc$ can be added to the subgoal. $Susp$ is a conjunction of suspended atoms; the equalities in the resulting implication may be able to instantiate atoms in $Susp$ such that they can be further unfolded.
Logical equivalence transformations include splitting (replace \((A \lor B) \land C\) by \((A \land C) \lor (B \land C)\)) and logical simplifications such as the replacement of \(A \land \text{false}\) by \text{false}.

Equality rewriting implements the unification algorithm to deal with equality atoms introduced into the goal. The used rewrite rules are adapted from [Fu96] where they were based on [MaMo82]. Conceptually, equality rewriting may also be regarded as propagation with the axioms of Clark's Equality Theory (CET, [Cl78]), followed by deletion of the no longer needed, rewritten terms.

If there is a derivation of \(G_n = D \lor \text{Rest}\) from an initial goal \(G_0\) and the operations of the proof procedure have been exhaustively applied to the disjunct \(D\) of the goal \(G_n\), then a computed answer to \(G_0\) is obtained from \(D\) by transforming all implications in \(D\) into denials (integrity constraints with \text{false} as the conclusion) which ensure that no further steps of propagation (with atoms not present in \(D\)) could generate new conclusions (see definition 3.1.7 for the precise definition).

Example 1.3.2 Suppose the goal disjunct \(D\) is

\[ p(a) \land q(a) \land [ p(x) \rightarrow q(x) ] \]

If further instances of \(p\) besides \(p(a)\) were true, then further instances of \(q\) would have to be true as well (but they might be defined as \text{false} in \(T\)). Therefore the implication is transformed into a denial and the computed answer obtained from \(D\) is

\[ p(a) \land q(a) \land [ p(x) \land x \neq a \rightarrow \text{false} ] \]

The operations of the proof procedure are all sound in the sense that they derive goals which are equivalent in \(T\), i.e.

\[ T \models_{\text{int}} \forall[G_i \leftrightarrow G_{i+1}]. \]

More general soundness results, some of which hold only with respect to one of the weaker answer notions mentioned above (see definition 2.3.2), can be obtained. Refutation soundness (ideal answers are never rejected) can be proven.
Completeness results (for every ideal answer there is a corresponding computed answer) are only obtainable for special cases and for extensions of the proof procedure.

Chapter 4: Unified Areas

The fourth chapter reviews related semantics and proof procedures previously defined for SQO, ALP, CLP, concurrent LP and concurrent CLP and compares them with the approach taken in the CALOG framework. The focus is on the first three, which are the main areas the framework seeks to unify, whereas the possibility of also incorporating concurrency has to be investigated further.

In the section on SQO the work done by Chakravarthy, Grant and Minker in [ChGrMi90, ChGrMi88] is discussed. Some of the restrictions imposed there have afterwards been lifted in [GaLo93]. It is shown that the CALOG framework fully encompasses SQO and that many of the special techniques used in query transformation procedures can be more easily and more generally performed by the CALOG proof procedure's operations.

In the case of ALP, where abducibles and equality are the only suspended predicates, conditions (A1)–(A3) of the CALOG framework's semantics can be directly mapped onto corresponding conditions in ALP semantics. Analogies between the propertyhood view of integrity constraint satisfaction and the generalized stable models introduced in [KaMa90] are pointed out. The CALOG proof procedure is compared in detail with the two closely related aforementioned abductive proof procedures on which it is partly based, the Iff Proof Procedure [Fu96, FuKo96] and SLDNFA [DeDS97, DeDS92].

Rather than employing a built-in constraint solver as most frameworks modelled after the CLP(X) scheme of [JaLa87] do, in the CALOG framework constraint definitions are augmented with explicit integrity constraints (to process suspended constraint atoms) which are similar to Constraint Handling Rules (CHRs) [Fr95]. Comparisons between CLP and standard LP have previously been conducted in [Ma92, Ma93] (see also [Ma87]). The thesis shows
how concepts of the CLP($X$) scheme and its semantics map to the CALOG framework and analyses the relationship with CHRs and other CLP extensions more closely.

The sections on concurrent LP and concurrent CLP are mainly meant as an incentive for future research. Having borrowed the concept of suspension from concurrent LP, the CALOG framework appears to have the potential to incorporate concurrency and might thus serve as a unifying platform for even more areas of Logic Programming.

**Chapter 5: Extensions**

An important extension of the proof procedure is propagation across disjunct boundaries (especially with respect to the second of the initial aims, i.e. to integrate global problem-solving strategies). For this purpose propagation is redefined in a way similar to non-clausal resolution [Mu82]. Since this usage of propagation is comparable to the handling of “disjunctive constraints”\(^1\) in CLP, the resulting methods have been called *Constraint Propagation with Disjunctions (CPD)*. The CPD methods constitute a generalization of several methods of handling disjunctive constraints in CLP, among them the forward-checking and look-ahead techniques of CHIP [VH89], Generalized Propagation [LPWa92] and Constructive Disjunction [VHSaDe93, JoSo93, Jo92]. They are also related to the Andorra principle [Wa88] and so-called “encapsulated search”, as in AKL [JaHa91] and Oz [ScSm94].

More technical improvements of the proof procedure are needed with respect to the efficiency of propagation and its termination. Since propagation has been defined as resolution, the problem of restricting propagation can be approached by looking at resolution restrictions proposed in the theorem-proving literature [ChLe73, EiOh93]. Possible restrictions include $P_1$-resolution, which corresponds to the requirement that the expression $\text{Susp}$ in the definition of propagation is empty, and hyper-resolution.

\(^1\)In the CALOG framework any disjunction can be regarded as a “disjunctive constraint”.

25
Another important extension of the proof procedure is the *deletion* operation. It interprets integrity constraints as identifying logically redundant and operationally useless subgoals which can be removed to simplify the goal. Ideally, deletion should be applied automatically by the proof procedure, e.g. the atom $X>0$ should be deleted as soon as the atom $X>1$ is added to the goal. This can be justified by the transitivity rule

$$X>Y \land Y>Z \rightarrow X>Z$$

but the same rule can also be applied by propagation in case the conclusion $X>Z$ is operationally useful. Criteria have to be worked out when integrity constraints are to be used for propagation and when for deletion.

### Chapter 6: Implementation and Applications

PROCALOG, short for *PROgramming with Constraints and Abducibles in LOGic* (or *PROgramming in CALOG*), is a realization of the CALOG framework and proof procedure as a programming language. It includes several of the extensions proposed in chapter 5.

The two currently existing prototype implementations are briefly presented and then used to provide some computational results for solutions to several example applications. Special emphasis is placed on assessing the practical effectiveness of the CPD extensions.

The framework and its implementations lend themselves to many well-known problems and applications of Artificial Intelligence (see e.g. [RuNo95, Wi92, Ko91]), especially problems with a large search space (where the CPD extensions can be used to drastically prune the search tree) and applications relying on knowledge representation in the form of condition-action (if-then) rules (where integrity constraints can often be used to model the rules in a logical way). The latter include expert systems for configuration problems or medical diagnosis.

Examples of constraint satisfaction problems (such as the $n$-queens problem or map colouring), Operations Research applications (such as job-shop
scheduling and warehouse location problems) and an approach to configuration which bears some similarity to the CALOG framework are discussed in detail.

The use of integrity constraints to improve the efficiency of logic (PROLOG) programs without having to abandon the original, natural specification is another interesting application of the framework.

Finally, a number of other possible applications from different areas of AI is mentioned and briefly discussed.

Chapter 7: Conclusion

The concluding chapter summarizes what has been achieved and points out what else can be done in future work.
Chapter 2

Unifying Framework and Semantics

CALOG is an acronym combining the most important ingredients of the framework: C for constraints, A for abducibles and LOG for logic. This chapter defines the CALOG framework and its semantics.

2.1 Knowledge representation

2.1.1 Definitions and integrity constraints

In the CALOG framework, knowledge is represented both by definitions and by integrity constraints.

The notion of atom is the same as in standard LP except that two special atoms, true and false, with the respective truth values, are introduced. Literals are atoms or their negations.

Definition 2.1.1 (Framework language)

- A definition is a formula
  
  \[ H \leftarrow D_1 \lor \ldots \lor D_n, \; n \geq 1 \]

  where \( H \) is an atom (but not true or false) and each \( D_i \) is a conjunction of literals. Variables in \( H \) are implicitly universally quantified with scope the entire definition whereas variables in a disjunct \( D_i \) which do not occur
in $H$ are existentially quantified with scope the disjunct. $H$ is called the head of the definition, $D_1 \lor \ldots \lor D_n$ is called the body.

The heads of two different definitions must not unify. A set of definitions is called a theory.

- An integrity constraint is a formula

$$\text{Cond} \rightarrow \text{Conc}$$

where $\text{Cond}$, the condition of the integrity constraint, is a conjunction of atoms and $\text{Conc}$, the conclusion of the integrity constraint, is a disjunction of atoms. Neither the condition nor the conclusion may be empty (but true and false are allowed). All variables are implicitly universally quantified with scope the entire formula.

Note that it is not necessary to allow negative literals in integrity constraints because negative condition (conclusion) literals can be turned into positive conclusion (condition) literals, yielding a logically equivalent formula with only positive literals.

In order to facilitate illustrating newly introduced concepts and putting them into context, one particular example, the $n$-queens problem, will be used repeatedly in the following. As a Constraint Satisfaction Problem (CSP) the $n$-queens problem at the same time highlights one important area of applications of the CALOG framework (cf. section 6.2).

Example 2.1.1 ($n$-queens problem: knowledge representation)

In the $n$-queens problem $n$ queens have to be placed on an $n \times n$ chessboard so that they do not attack each other. Two queens placed at coordinates $(A, B)$ respectively $(C, D)$ of the board attack each other if the rules of chess allow a queen to move from $(A, B)$ to $(C, D)$. This is the core of the problem specification and it can be directly expressed by a single integrity constraint generating false if the specification is violated:
queen(A, B) ∧ queen(C, D) ∧ move((A, B), (C, D)) → false

Here queen gives the positions of the queens on the chessboard as (Row, Column) tuples. The definition of move represents legal queen moves, e.g.

\[
\text{move}((A, B), (C, D)) \leftrightarrow \\
\quad ( A=C ∧ D∈[1, n] ∧ B≠D ) ∨ \\
\quad ( B=D ∧ C∈[1, n] ∧ A≠C ) ∨ \\
\quad ( \text{Diag}∈[-n, n] ∧ \text{Diag}≠0 ∧ 0<A+\text{Diag}≤n ∧ 0<B+\text{Diag}≤n ) ∨ \\
\quad ( \text{Diag}∈[-n, n] ∧ \text{Diag}≠0 ∧ 0<A+\text{Diag}≤n ∧ 0<B-\text{Diag}≤n )
\]

where \( X∈[M, N] \) holds whenever \( X \) is an integer with \( M<X<N \) (the definition of "∈" may be added explicitly or assumed to be built-in, see below).

For efficiency purposes and also for other reasons (see section 2.2), it may be preferable to express the definition of move as a list of ground facts:

\[
\text{move}((1, 1), (1, 2)) \leftrightarrow \text{true} \\
\text{move}((1, 1), (1, 3)) \leftrightarrow \text{true} \\
\ldots
\]

This can also be achieved by precompiling the previous definition of move for a given value of \( n \).

Note that, unlike in most solutions to the \( n \)-queens problem, the definition of move is not interwoven with the algorithm to solve the problem. Arguably, the CALOG framework thus allows a neater separation of factual and problem solving knowledge.

In standard LP, definitions are written in if form and a set of such definitions constitutes a normal logic program, whereas here definitions are written in if-and-only-if form and with disjunctions in the bodies. However, there is an obvious link between normal logic programs and theories of if-and-only-if definitions.

**Definition 2.1.2** If a theory \( T \) of if-and-only-if definitions is transformed into a set of if clauses (i.e. into a normal logic program) \( \mathcal{P} \) by rewriting a definition
$H \leftrightarrow D_1 \lor \ldots \lor D_n$, $n \geq 1$

into the $n$ clauses

\[ H \leftarrow D_1 \]
\[ \ldots \]
\[ H \leftarrow D_n \]

then $\mathcal{P}$ is the normal logic program corresponding to $T$.

The same normal logic program $\mathcal{P}$ may correspond to different theories which differ in the extent to which their definitions are homogenized in the usual sense:

**Definition 2.1.3 (Homogenized form)**

A definition is in homogenized form iff all terms in the head are mutually distinct variables.

In the Clark completion, $\text{Comp}(\mathcal{P})$, of a normal logic program $\mathcal{P}$, all definitions are written in homogenized form and there is exactly one definition per predicate (cf. [Ho90]). It is straightforward to extend the notion of Clark's completion to a theory $T$ of definitions: to obtain $\text{Comp}(T)$, homogenize every definition in $T$ and then create one homogenized definition for every predicate by combining the bodies of all definitions for the same predicate into one disjunction.

**Lemma 2.1.1** If the normal logic program $\mathcal{P}$ corresponds to the theory $T$, then, up to renaming of variables,

\[ \text{Comp}(T) = \text{Comp}(\mathcal{P}) \]

**Proof:** Clear from construction. 

Note that completing $T$ adds new information as there is no requirement in the framework that all predicates be completely defined. In particular, predicates which have no definitions, are defined as false for all their instances in the completion.
If there is no negation in the bodies of definitions in \( T \), then the corresponding logic program \( P \) is a Horn clause theory and \( T \) and \( P \) have the same minimal Herbrand model. Otherwise lemma 2.1.1 shows that \( T \) and \( P \) have the same models under the 2-valued completion semantics (since the models under this semantics are the models of the completion), but not necessarily under other semantics (cf. section 2.5).

The chief reason for the use of if-and-only-if definitions instead of if clauses is to ensure that both the unfolding and the propagation operation of the proof procedure (chapter 3) are sound. Whereas standard LP negates the initial goal to enable resolution with the heads of Horn clauses, here the initial goal is not negated, to allow propagation with integrity constraints. However, replacing atoms in the goal by their definitions (unfolding) still has to be possible and for this operation to be sound the only-if halves are needed.

**Example 2.1.2** (Intended execution of definitions and integrity constraints)

\[
\begin{align*}
T_u & : p \leftrightarrow r \\
& \quad q \leftrightarrow s \\
IC & : r \land q \rightarrow \text{false} \\
G_0 & : p \land s
\end{align*}
\]

These four sentences together imply \( \text{false} \). To derive \( \text{false}, p \) in \( G_0 \) may be replaced by \( r \) (using the only-if half of the first definition in \( T_u \)), yielding

\[ r \land s \]

and \( q \) in the integrity constraint may be replaced by \( s \), yielding

\[ r \land s \rightarrow \text{false} \]

The results of the replacements can be used to derive (propagate) \( \text{false} \) by two steps of resolution (or one step of hyper-resolution).

Note that \( \text{false} \) would not have been logically implied if the definitions in \( T_u \) had been written in if form (but the minimal model of the corresponding set of Horn clauses is empty). See also example 3.3.2.
2.1.2 Predicates

Definition 2.1.4 (Predicate types)

There are three kinds of predicates in the CALOG framework:

- **user-defined** predicates, defined by a theory $T_u$ of user-provided definitions,
- **built-in** predicates, defined by a theory $T_b$ of built-in definitions,
- **external** predicates, defined by an inaccessible or unknown theory $T_e$ of definitions.

The symbol $T$ is used to denote the union of the three sub-theories:

$$T = T_u \cup T_b \cup T_e$$

and $T'$ refers to the accessible parts of the theory:

$$T' = T_u \cup T_b$$

Definitions in $T'$ are called accessible, definitions in $T_e$ are called inaccessible.

Example 2.1.3 (n-queens problem: predicate types)

In example 2.1.1, move is a user-defined predicate whereas queen is an external predicate whose definition is initially unknown. Possible definitions (corresponding to solutions of the n-queens problem) are to be discovered by the program. The predicate "E" used in the (first) definition of move may be available as a built-in predicate (or otherwise it also has to be user-defined).

Built-in predicates, which correspond to constraint predicates in CLP, always include equality and generally also arithmetic inequality ($<$, $\leq$, $\geq$, $>$) and arithmetic operators (like plus). Further predicates may be added in an actual implementation. Conceptually, the main difference between built-in predicates and user-defined predicates is that the former have fixed definitions, which the user cannot change.
Moreover, it will be assumed that the definition of a built-in predicate is of a specific format.

**Definition 2.1.5 (Form of built-in definitions)**

If \( p \) is a built-in predicate, then its definition is of the form

\[
p(t_1, \ldots, t_m, x_1, \ldots, x_n) \iff x_1 = c_1 \land \ldots \land x_n = c_n
\]

where \( m, n \geq 0 \), \( m + n > 0 \) and \( t_1, \ldots, t_m \) are ground terms. Thus the values of the output parameters \( x_1, \ldots, x_n \) are uniquely determined if all input parameters \( t_1, \ldots, t_m \) are ground.

If \( p \) is equality or inequality, then there are no output parameters, and so the definition is an enumeration of all ground facts (0=0, 0<1, etc.). The standard arithmetic operators have two input and one output parameter, e.g. \( \text{plus}(1, 2, x) \iff x=3 \). This has the consequence that \( \text{plus} \) can only be used for additions and not for subtractions, e.g. \( \text{plus}(1, X, 3) \) cannot be reduced to \( X=2 \). To achieve a better behaviour (where \( \text{plus}(1, X, 3) \) is reducible), one could relax the framework restriction that heads of different definitions in \( T \) must not unify — they might be allowed to unify as long as the respective definitions do not contradict each other. Then there could also be a definition \( \text{plus}(1, X, 3) \iff X=2 \). On the other hand, this is really a definition for the \( \text{minus} \) predicate. The user can circumvent the problem by providing a user-defined predicate, say \( \text{sum} \), which is defined in terms of \( \text{plus} \) and \( \text{minus} \) depending on which parameters are instantiated.

External predicates correspond to abducibles in ALP and extensional predicates in SQO. To regard \( T_e \) as inaccessible rather than generally non-existent is motivated by the SQO case where the extensional database (EDB) is given but not used during the query transformation. However, external predicates do not have to have definitions — just as some instances of user-defined predicates may be left undefined, so may some or all instances of external predicates. See also the remark after lemma 2.1.1.

While the set of built-in predicates has to be disjoint from the other two
sets of predicates, it is not necessary to require that the sets of user-defined and external predicates be disjoint. So the user can supply partial definitions of predicates which may have additional true instances which are (conceptually) defined externally and may be abduced (generated in the course of the computation). Alternatively, a similar effect can be achieved by supplying only integrity constraints and no definitions, e.g. if p is an external predicate, then

\[
\begin{align*}
\text{true} & \rightarrow p(1) \\
p(2) & \rightarrow \text{false}
\end{align*}
\]

are two integrity constraints stating that p is true for 1 and false for 2; further instances of p may be abducible.

### 2.2 Computational task

The computational task is formalized by the declarative semantics (specifying what kind of answer is to be computed) and by the proof procedure (specifying how these answers are to be computed) in section 2.3 and in chapter 3. The informal description of the computational task given in this section is to motivate some of the concepts needed in the formal definition of the declarative semantics and the proof procedure.

So, informally, the computational task is to reduce a given initial goal to a disjunction of (computed) answers which is equivalent to the initial goal in the sense that the goal and the disjunction are satisfied, in \( T \), by the same assignments to the free variables. Such answers are obtained by using the accessible definitions in \( T' \) to reduce (also called "unfold") atomic goals to equivalent goals in the form of disjunctions. Integrity constraints are used to check consistency of answers with the whole theory \( T \) (including the inaccessible part \( T_e \)).

#### 2.2.1 Goals

**Definition 2.2.1** An *initial goal* is a conjunction of literals with all variables
The initial goal is specified by the user. Subsequent goals can be defined operationally as the formulas generated, in one or more steps, by the operations of the proof procedure. Those operations may introduce disjunctions and implications into the goal, i.e. formulas of the same form, $\text{Cond} \rightarrow \text{Conc}$, as integrity constraints. All atoms in a goal which are not part of an implication will also be referred to as **atomic subgoals**.

**Example 2.2.1 (n-queens problem: goals)**

For $n = 3$ a possible initial goal is

$$\text{place}(1) \land \text{place}(2) \land \text{place}(3)$$

where \text{place} is user-defined by the definition

$$\text{place}(\text{Col}) \leftrightarrow \text{queen}(\text{Col}, 1) \lor \text{queen}(\text{Col}, 2) \lor \text{queen}(\text{Col}, 3)$$

Unfolding the initial goal introduces three disjunctions into the goal. Splitting disjunctions ultimately enables propagation with the integrity constraint from example 2.1.1 to generate \text{false} in every disjunct, proving that the 3-queens problem has no solution. The n-queens problem will also be used in chapter 5 to illustrate how a generalization of propagation can be used to reject unsatisfiable disjuncts as soon as possible, sometimes without any splitting.

Unless otherwise indicated by the use of explicit quantifiers, the following variable quantification conventions will be applied to goals:

- all variables in an initial goal are free (as imposed by the definition)
- all variables in goal literals which do not appear in the initial goal are existentially quantified (with scope the entire goal)
- all remaining variables (i.e. variables in implications which neither appear in the initial goal nor in a goal literal) are universally quantified with scope the implication in which they occur
Example 2.2.2 If the initial goal is \( p(X) \), then the goal
\[
p(X) \land q(X, Y) \land [ r(X, Y, Z) \rightarrow \text{false} ]
\]
is short for the formula
\[
\exists Y \ ( p(X) \land q(X, Y) \land \forall Z [ r(X, Y, Z) \rightarrow \text{false} ] )
\]

If certain restrictions are imposed on definitions and integrity constraints, then explicit quantifiers are never needed. For example, in [Fu96, FuKo96] definitions and integrity constraints (and also initial goals) have to be range-restricted, i.e. all variables have to appear in positive body or condition literals (see section 3.1.3 for the precise definition and a more detailed discussion of the issue). Without such restrictions, universally quantified variables may appear in atomic subgoals during the computation. Following [DeDS97] the proof procedure presented in the next chapter relies instead on a safety restriction when selecting atoms (similar conceptually to SLDNF, which flounders if all literals in a goal are non-ground and negative).

Example 2.2.3
\[
\mathcal{T}_u : p(X) \leftrightarrow \text{true}
\]
\[
\mathcal{IC} : p(X) \rightarrow q(X)
\]
The atom \( p(X) \) in the integrity constraint can be reduced to \text{true}. If the resulting implication
\[
\text{true} \rightarrow q(X)
\]
is simplified to
\[
\forall X \ q(X)
\]
then a goal atom with a universally quantified variable is generated (which may cause problems further discussed in section 3.1.3).

2.2.2 Suspension
An atom \( A \) cannot be reduced ("unfolded") if its definition is inaccessible (i.e. if \( A \) is an atom of an external predicate) or, in general, unknown. Moreover, to control the amount of search non-determinism, \( A \) shall not be reduced if its
predicate has accessible definitions, but variables in $A$ have to be instantiated to unify with the head of one or more definitions. If reducing $A$ is desired in this case, then the definitions could be rewritten accordingly (cf. below after the examples).

**Definition 2.2.2** An atom $A$ is **reducible** if there is an accessible definition $H \leftrightarrow D_1 \lor \ldots \lor D_n$ in $T'$ such that $A$ is an instance of $H$ (i.e. $A = H\sigma$ for some substitution $\sigma$). Otherwise $A$ is **suspended**.

Note that, since the heads of two definitions must not unify, there is one and only one definition for $A$ if $A$ is reducible. Even if this restriction was relaxed in the way discussed after definition 2.1.5 it would not matter which of several applicable definitions was used as they would still be required to yield the same result.

The notion of suspension provides the basis for a uniform treatment of user-defined, built-in (constraint) and external (abducible or EDB) predicates. Answers will not be defined in terms of literals of either predicate category but rather in terms of suspended literals, see definition 2.3.2.

**Example 2.2.4** (Suspension of user-defined predicates)
Let the user-defined predicate $p$ be defined by

$p(X, a) \leftrightarrow q(X)$

$p(X, b) \leftrightarrow r(X)$

Then $p(X, t)$ and $p(a, b)$ are reducible (and can be reduced to $q(X)$ and $r(a)$ respectively), whereas $p(X, Y)$ and $p(a, Y)$ are suspended. If the two given definitions are the only definitions for the predicate $p$, then $p(X, c)$ is also suspended. If the user wants $p(X, c)$ to generate $\text{false}$, an explicit definition $p(X, c) \leftrightarrow \text{false}$ or an integrity constraint $p(X, c) \rightarrow \text{false}$ should be added. Otherwise $p$ might be regarded as only partially (user-)defined, with further instances like $p(a, c)$ being "abducible".

If the user wants no $p$-atom to be suspended, then the definition of $p$ could be rewritten to
\[ p(X,Y) \iff (q(X) \land Y=a) \lor (r(X) \land Y=b) \]

In an implementation it may be convenient to allow the user to specify instead that an atom of a user-defined predicate is to be reduced to \texttt{false} whenever it does not unify with any of the given definitions for its predicate.

**Example 2.2.5** (Suspension of built-in predicates)

According to the way definitions of built-in predicates are assumed to be written (see previous section), the atoms \( X<Y \), \( X>0 \) and \( \text{plus}(2,X,Y) \) are all suspended, but \( 1>0 \) and \( \text{plus}(2,2,X) \) are reducible (to \texttt{true} and \( X=4 \), respectively).

Atoms like \( 1<0 \), which are false in the minimal Herbrand model of \( T_0 \), are suspended, but can generate \texttt{false} if appropriate integrity constraints are used (e.g. \( X<Y \land Y<X \rightarrow \texttt{false} \)). Alternatively, one might extend the built-in theory \( T_0 \) to include definitions like \( 1<0 \iff \texttt{false} \) or a generalization thereof, cf. the footnote to the previous example.

**Example 2.2.6** (Suspension of external predicates)

External predicates are always suspended. Atoms of external predicates, which appear in the initial goal or are introduced into the goal by reducing user-defined atoms or by resolution with implications, will forever remain irreducible in the goal, although their variables may be instantiated or constrained by equalities in the course of the computation.

Just like suspended atoms of the other two types of predicates, they may be used for resolution with implications which may lead to \texttt{false} being generated.

Suspension, either defined similarly to definition 2.2.2 or according to user-defined suspension rules, is an important feature of concurrent LP languages [Sh89, Ja91] (see also section 4.4.1) and concurrent CLP languages [ScSm94, SaRi90] (section 4.4.2) where it is often used to eliminate search non-determinism entirely. In the CALOG framework, the user controls suspension and the extent of search non-determinism by controlling the form in which the definitions of user-defined predicates are written.
The user can control which atoms of user-defined predicates are suspended by writing definitions in more or less homogenized form: if a predicate is defined by an enumeration of all ground facts, then any of its atoms with variables in it is suspended, whereas atoms of predicates which have only one completely homogenized definition are never suspended.

The concept of suspension is also related to delay mechanisms first employed in MU-PROLOG [Na82]. In this and several other PROLOG dialects (including CHIP [VH89]) a control statement of the form

\[ \text{delay } p(a_1, \ldots, a_n) \]

where each \( a_i \) is either a "+" or a "-", specifies that an atom \( p(t_1, \ldots, t_n) \) can be selected only if all its arguments corresponding to a "+" in the delay declaration are ground. Suspension as used in concurrent LP languages and in the CALOG framework is a more general concept and does not require any explicit control.

**Example 2.2.7 (n-queens problem: suspension)**

Atoms of the external predicate queen are always suspended. Whether and which atoms of the user-defined predicate move are suspended depends on which of the two alternative definitions is used. The first definition of example 2.1.1 is in homogenized form, so atoms of move are never suspended if this definition is used. This may have the disadvantage that the move atom in the integrity constraint

\[ \text{queen}(A, B) \land \text{queen}(C, D) \land \text{move}((A, B), (C, D)) \rightarrow \text{false} \]

may be reduced without any of its variables being instantiated, which is probably not intended. If, instead, the second definition, which is a list of all ground facts, is used, then the move atom is suspended until all of its variables become instantiated.

Note that it is also possible to write the definition of move in such a way that \( \text{move}((A, B), (C, D)) \) is suspended iff the first two variables are uninstantiated. In practice this may result in yet another operational behaviour which might be better represented by the logically equivalent integrity constraint
queen(A, B) \land \text{move}((A, B), (C, D)) \land \text{queen}(C, D) \rightarrow \text{false}

Using this integrity constraint and assuming that the condition atoms are evaluated from left to right, the definition of move could again be in homogenized form.

Thus different algorithms can be simulated by different formulations of definitions, combined with control rules affecting the order of execution. See also example 5.2.2 and section 5.1.3.

Suspension can also be used to stop the computation at a point where the derived answer is more informative than it would be if it were further reduced.

Example 2.2.8 Consider an employee database and a query asking for employees X who are entitled to certain benefits such as bonus payments or maternity leave. Returning \text{manager}(X) or \text{female}(X) as an answer to such a query may be more informative than a disjunction of all managers or female employees. Moreover, such intensional answers remain correct even if the database is changed.

Writing down the definitions of the predicates \text{manager} and \text{female} as a set of ground facts results in the desired behaviour. The user-defined predicates, which are, from the deductive database and SQO point of view, EDB predicates, are suspended as long as their arguments are not ground. Only if the arguments are ground are the definitions accessed.

Suspended atoms cannot be reduced by means of definitions in \( T \) and are processed instead by means of integrity constraints. This processing may add to the goal, in one or more steps, either logically redundant (surrogate) subgoals some of which may be reducible and thus enable the reduction using definitions to proceed, or \text{false} which signals that the solution can be rejected as unsatisfiable.

Therefore, integrity constraints are used as approximations of the definitions in \( T \). To guarantee that they are sound approximations, integrity constraints will be required to be \text{properties} of the definitions, as defined below in
2.2.3 Framework instances

The definition of suspension from the previous section is very similar to the definition given in [Sh89] for concurrent LP languages. A more detailed comparison with concurrent LP can be found in chapter 4 which also compares the semantics and proof procedure of the CALOG framework with those of the main framework instances: standard LP, ALP, CLP and SQO. Here only a link between the CALOG framework and these four instances is established in order to put into context the concepts on which the framework builds.

In all of the four main framework instances there is no suspension of user-defined predicates. This corresponds to the case where all user-defined definitions are written in homogenized form.

In standard LP the only built-in predicate is equality, and there are no integrity constraints.\(^1\) Since (non-ground) equality atoms are the only suspended atoms, the computational task is to find answers to the initial goal, in the form of a conjunction of equalities assigning values to the free variables in the initial goal.

In ALP, the external predicates are abducible, and equality is the only built-in predicate. Since abducibles and (non-ground) equality atoms are the only suspended atoms, the computational task is to find an explanation for the initial goal in the form of a conjunction of abducibles and equalities which implies the initial goal and satisfies the integrity constraints.

In SQO, the external predicates are extensional database (EDB) predicates. For the purposes of query optimization, their definitions are considered inaccessible. The user-defined predicates are intensional database predicates, and integrity constraints approximate the extensional part. The only built-in

\(^1\) However, one may think of unification as a form of propagation with the integrity constraints being axioms of the Clark Equality Theory (CET), cf. section 2.4.3.
predicate is equality. The computational task is to reduce the initial query to an optimized query involving extensional predicates only. The integrity constraints eliminate (some) queries which cannot be satisfied by the EDB. Since the EDB is inaccessible and the integrity constraints only approximate the EDB, an answer thus obtained (in the form of an optimized query) only approximates the answers obtainable by submitting the optimized query to the EDB.

In CLP there are no external predicates. Built-in predicates are called constraints. Rather than employing a built-in constraint solver, as in most frameworks modelled after the $CLP(X)$ scheme of [JaLa87], the constraint solver is programmed by means of built-in constraint definitions augmented with explicit integrity constraints (to process suspended constraint atoms). This use of integrity constraints is similar to Frühwirth's constraint handling rules [Fr95]. Because integrity constraints can contain both user-defined and built-in predicates, user-defined predicates may be regarded as user-defined constraints. In principle it is also possible to integrate, for efficiency reasons, built-in constraint solvers into the CALOG framework; conceptually they correspond to built-in definitions together with integrity constraints. The treatment of equality in the proof procedure (see definition 3.1.4) may be understood in this way.

**Example 2.2.9** (Configuration of a computer system viewed in the framework instances ALP, CLP, SQO)

$T_0 : \text{processor}(X) \iff (X = \text{pentium}) \lor (X = \text{sparc}) \lor \ldots$

$\text{op-sys}(X) \iff (X = \text{os2}) \lor (X = \text{unix}) \lor \ldots$

$I_C : \text{processor}(\text{sparc}) \rightarrow \text{op-sys}(\text{unix})$

$\text{processor}(\text{sparc}) \land \text{op-sys}(\text{os2}) \rightarrow \text{false}$

CLP instance: The definitions of processor and op-sys may be given in $T_0$. The query $\text{processor}(X) \land \text{op-sys}(\text{os2})$ derives the constraint $X=\text{pentium}$ as an answer (by reducing processor to a disjunction and then checking consistency with $I_C$, thereby rejecting the disjunct with $X=\text{sparc}$).
SQO instance: The definitions of processor and op_sys may be contained in the extensional database (i.e. in $T_e$) and regarded as inaccessible during query optimization. The query $\text{processor(sparc)} \land \text{op_sys(os2)}$ can be rejected as inconsistent with $IC$ without accessing $T_e$.

ALP instance: The definitions of processor and op_sys may be regarded as unknown (so again part of $T_e$), and the two predicates treated as abducibles. Given a query $\text{processor(sparc)}$, the answer $\text{processor(sparc)} \land \text{op_sys(unix)}$ is an abductive explanation which satisfies the integrity constraints (whereas an answer containing $\text{op_sys(os2)}$ does not).

2.3 Declarative semantics

2.3.1 Intended models

When specifying the declarative semantics for a new framework, authors usually either adapt a particular notion of model previously proposed for normal logic programs (such as stable, perfect, or well-founded model — cf. section 2.5) or they invent a new notion. To achieve greater flexibility, the CALOG framework will leave the choice of model to the user: for every theory $T$, the user has to identify the intended models of $T$.

In order to achieve meaningful results (for the semantics and when applying the proof procedure), the choice of intended models for a given theory $T$ must satisfy the following requirements:

- every intended model of $T$ must be a (2- or 3-valued) Herbrand model of $T$
- $T$ must have at least one intended model
- if $M$ and $M'$ are models of $T$ and $M' \subset M$ and $M$ is an intended model of $T$, then $M'$ is also an intended model of $T$
- the interpretation of built-in predicates is the same in all intended models
If only 2-valued interpretations are considered, then the second requirement implies that $T$ has to be consistent in the classical sense (i.e. it has a 2-valued model), which may not be easy to verify. To avoid this problem, 3-valued intended models can be used (e.g. the definition $p \leftrightarrow \neg p$ is true in an intended model in which the truth value of $p$ is undefined, cf. section 2.5).

The third requirement ensures that the unique minimal Herbrand model of a negation-free theory $T$, which corresponds to a Horn-clause program, is always an intended model of $T$. Together with the last requirement it implies that the interpretation of built-in predicates is always the same as that in the minimal Herbrand model. This is important for the treatment of equality, see the next section.

The next section also presents a new view of integrity constraint satisfaction which is based on the notions of intended model and the derived notion of property:

**Definition 2.3.1** A first-order sentence $\Phi$ is a property of a set of first-order sentences $T$ iff $\Phi$ is true in all intended models of $T$, written $T \models_{\text{int}} \Phi$. If $P$ is a set of sentences, $T \models_{\text{int}} P$ denotes that every sentence in $P$ is a property of $T$.

Since a theory in the CALOG framework must have at least one intended model, the case where every sentence is a trivial property of a theory without intended models is ruled out.

### 2.3.2 Answer definition(s)

As the CALOG framework aims to unify various different instances, it appears sensible to define a rather flexible declarative semantics with several answer notions: weak, good, and ideal. Answers computed by the proof procedure in the general case will have to satisfy at least the definition of weak answer (affirmation soundness) and the proof procedure must never reject ideal answers.
(refutation soundness). In some special cases or under certain additional assumptions (e.g. on the integrity constraints) computed answers can be shown to be good or even ideal answers (see section 3.3).

Definition 2.3.2 A is an ideal answer to the initial goal $G_0$ if

(A1) $A$ is a conjunction of atoms and implications with all atoms being suspended (including those occurring in implications)

(A2) $T \models_{in} \forall [G_0 \leftrightarrow A \lor Rest]$ for some formula $Rest$

(A3i) $T \not\models_{int} \exists \exists A$

$A$ is a good answer to $G_0$ if (A1) and (A2) hold and

(A3g) $T \cup IC \cup CET \cup \exists \exists A$ is consistent, where $CET$ denotes the Clark Equality Theory [Cl78]

$A$ is a weak answer to $G_0$ if (A1) and (A2) hold and

(A3w) $IC \cup CET \cup \exists \exists A$ is consistent

Any ideal, good or weak answer to $G_0$ is also referred to as a semantic answer to $G_0$.

Condition (A1) makes explicit the requirement that goals should be reduced as far as possible, i.e. until all atoms are irreducible and thus suspended. In several special cases all implications can be disposed of so that $A$ is a conjunction of atoms only. Condition (A1) generalizes the requirements of $A$ being a conjunction of equalities in the LP case, a conjunction of abducibles in the ALP case, a conjunction of EDB atoms in the SQO case, and a conjunction of constraints in the CLP case. If $A$ contains implications, they will usually be written in the form of denial integrity constraints (implications with false as the conclusion), see chapter 3.
Condition (A2) means that, in the intended models of $T$, $G_0$ and $A \lor Rest$ are true for the same assignments to the free variables in $G_0$, i.e. $G_0$ and $A \lor Rest$ are equivalent in the intended models of $T$. The use of the universal closure relaxes the restriction in ALP that abductive explanations contain no variables.

There is another formulation of condition (A2) which more closely resembles analogous requirements imposed on answers in ALP and CLP (cf. sections 4.2 and 4.3):

\[(A2') \; T \models_{int} \forall[G_0 \leftarrow A]\]

The following lemma justifies using conditions (A2) and (A2') interchangeably.

Lemma 2.3.1 Conditions (A2) and (A2') are equivalent, i.e. any answer $A$ which satisfies (A2) also satisfies (A2') and vice versa.

Proof:

(A2) $\Rightarrow$ (A2'): Regardless of what $Rest$ is, $G_0 \leftarrow A \lor Rest$ always implies $G_0 \leftarrow A$.

(A2') $\Rightarrow$ (A2): It is sufficient to specify a $Rest$ for which condition (A2) holds. With $Rest = \neg A \land G_0$, it remains to be shown that $G_0 \leftarrow A \lor G_0$. This follows from the assumption $G_0 \leftarrow A$ together with the two trivial implications $G_0 \leftarrow G_0$ and $G_0 \rightarrow A \lor G_0$. 

Conditions (A1) and (A2) imply that answers may be obtained from a goal which is equivalent to the initial goal (in the intended models of $T$), but which has been simplified and transformed to a disjunction of answers all of whose atoms are irreducible (cf. the initial formulation of the computational task in section 2.2). This might be formalized by requiring that

\[T \models_{int} \forall[G_0 \leftarrow A_1 \lor \ldots \lor A_n]\]

where $A_1, \ldots, A_n$ are precisely all the answers to $G_0$ (which also satisfy (A1))
and one of the (A3) variants). This requirement is stronger than (A2) and no longer equivalent to (A2') as it does not depend on a formula Rest which is left unspecified. Unfortunately, it rules out the possibility of Go having infinitely many answers (which (A2) does not).

In addition to (A1) and (A2) a third condition is needed to guarantee that the answer is meaningful in some sense, ideally that it is satisfiable in at least one intended model of T. This is what (A3,) says: if the negation of \( \exists A \) is not a property of T, then there is an assignment to the free variables in A which makes A true in at least one intended model of T. It follows that, in the special cases in which there is only one intended model, condition (A3,) is equivalent to

\[ T \models_{int} \exists A. \]

For example, this is the case if T corresponds to a Horn clause theory or is (locally) stratified and the intended model is the unique minimal Herbrand model (which coincides with the unique perfect, stable and well-founded model; see also section 2.5 for a more detailed discussion).

The different variants of the third condition are introduced because the suspension of subgoals restricts the extent to which the satisfiability of a potential answer can be determined. In the case of good answers, condition (A3,\( _g \)) insists on consistency with the accessible part \( T' \) of the theory and uses integrity constraints to ensure consistency with whatever is known about the inaccessible part \( T_e \) of the theory.

The cases of ALP and SQO make it clear that it is not possible to insist that answers always be ideal: \( T_e \) is unknown in ALP, and in SQO it is not expected that the final transformed query is always satisfiable in the EDB. In both cases it is, however, reasonable to insist that good answers be computed (i.e. that condition (A3,\( _g \)) is satisfied).

Condition (A3,\( _w \)) only demands consistency with the integrity constraints (again including the CET axioms as the “integrity constraints” for equality). This notion of weak answer is introduced because suspension may effectively
make parts of $T'$ inaccessible: for example, in the CLP case, where the satisfiability check would usually be performed by a built-in constraint solver with respect to a given constraint domain, and in the general case, where literals of user-defined and built-in predicates may also be suspended and may prevent the proof procedure from checking the consistency of suspended subgoals with $T'$. For example, if

$$x > 0 \rightarrow \text{false}$$

occurs in a goal and $x$ is universally quantified over the implication, then the goal is inconsistent with $T$, but $x > 0$ is suspended and the inconsistency may not be detected.

Obviously, every good answer is a weak answer, and $T \models_{\text{int}} I \cup C \cup E T$ (see next section) ensures that every ideal answer is a good answer. The following example shows that the converse does not hold in either case.

Example 2.3.1 (Difference between answer notions)

If $I \cup C$ is empty, then

$$G_0 : 1 < x \land x < 0$$

is a good answer to itself, as there are (non-intended) models of $T$ in which $G_0$ is true, e.g. any model in which $2 < 0$ holds. But $G_0$ is not an ideal answer, as $G_0$ is false in the minimal Herbrand model of the built-in definition of "$<$" (in which "false" inequalities like $2 < 0$ do not hold).

If $I \cup C$ contains the transitivity and irreflexivity rules from Example 2.4.2, then $G_0$ is no longer a good answer (because $G_0$, together with transitivity, implies $1 < 0$ which, together with $0 < 1$ from the theory $T'$, implies $1 < 1$ which, together with irreflexivity, yields a contradiction). But $G_0$ is still a weak answer (because $0 < 1$ is not implied by $I \cup C$ alone).

The example illustrates that the consistency check required for good answers does not ensure satisfiability in the intended models.
2.4 Integrity constraint satisfaction

The integrity constraints $\mathcal{IC}$ should be thought of as approximations of the (intended meaning of) definitions in $\mathcal{T}$. The proof procedure checks possible answers to an initial query for satisfaction of the integrity constraints.

2.4.1 The propertyhood view

The two simplest views of integrity constraint satisfaction are the theoremhood view, requiring the integrity constraints to be true in all (classical) models of $\mathcal{T}$, i.e. $\mathcal{T} \models \mathcal{IC}$, and the consistency view, requiring the integrity constraints to be true in just one model of $\mathcal{T}$, i.e. $\mathcal{T} \cup \mathcal{IC}$ is consistent.

Neither of these two views takes into account the distinction between those models of $\mathcal{T}$ which are actually intended and those which are not. This thesis proposes an alternative view of integrity constraint satisfaction which requires the integrity constraints to be true in the intended models of the theory:

**Definition 2.4.1** If $\mathcal{T}$ is a set of definitions and $\mathcal{IC}$ a set of integrity constraints such that $\mathcal{T} \models_{\text{int}} \mathcal{IC}$, then $\mathcal{T}$ is said to satisfy $\mathcal{IC}$ according to the propertyhood view of integrity constraint satisfaction.

Recall that $\mathcal{T} = \mathcal{T}' \cup \mathcal{T}_e$ where $\mathcal{T}_e$ is the inaccessible part of the theory $\mathcal{T}$. Consequently, the propertyhood view can be equivalently formulated as

$\mathcal{T}' \cup \mathcal{T}_e \models_{\text{int}} \mathcal{IC}$

which more closely resembles the usual formulations of integrity constraint satisfaction in ALP, e.g. the theoremhood view

$\mathcal{T} \cup A \models \mathcal{IC}$

where $A$ is an answer to the initial goal (usually the completion of the answer is used; cf. [Fu96]). The alternative formulation of the propertyhood view also corresponds to an interpretation of the computational task in the CALOG framework which is different from the one described in section 2.2: to find candidate theories $\mathcal{T}_e^*$ such that
$T' \cup T^* \models_{int} IC$

holds (see theorem 3.3.4 and also section 4.2).

### 2.4.2 Comparison with other views of IC satisfaction

If a sentence is true in all models of a theory $T$, then it is true in all its intended models. Thus sentences which are logically implied by $T$ are also properties of $T$ ("$\models$" $\Rightarrow$ "$\models_{int}$"), i.e. integrity constraints which are satisfied according to the theoremhood view are also satisfied according to the propertyhood view, but not vice versa.

If a sentence is a property of a consistent theory, then the sentence is also consistent with the theory (note that this depends on the assumption that the theory does have intended models). Thus integrity constraints which are satisfied according to the propertyhood view are also satisfied according to the consistency view.

**Example 2.4.1**

Let $T: p \leftrightarrow p$

Assume that the only intended model is the unique minimal Herbrand model of $T$ (which is empty). $T$ also has a (non-minimal) model in which $p$ is true. Hence the integrity constraint

$$p \rightarrow \text{false}$$

is satisfied according to the consistency and propertyhood views, but not according to the theoremhood view, whereas the integrity constraint

$$\text{true} \rightarrow p$$

is satisfied according to the consistency view, but not according to the propertyhood and theoremhood views.

The example may be compared to example 3.7 of [Fu96] which was presented there as an argument in favour of the theoremhood view. As the example shows, the propertyhood view agrees with the theoremhood view on whether the positive assertion $\text{true} \rightarrow p$ is satisfied. However, unlike the theoremhood view which also rejects $p \rightarrow \text{false}$, the propertyhood view is
asymmetric and prefers negative assertions in a similar way as negation by failure: something is assumed to be false if there is no reason to believe it is true. Note that this holds only for the case where the unique minimal Herbrand model is the single intended model of a Horn clause theory.

**Example 2.4.2** The following sentences are all properties of the built-in theory $T$ (whose interpretation is fixed in all intended models) and are therefore integrity constraints satisfied by any theory $T$ according to the propertyhood view.

\[
\begin{align*}
X < X & \rightarrow \text{false} \quad \text{(irreflexivity)} \\
X < Y \land Y < X & \rightarrow \text{false} \quad \text{(anti-symmetry)} \\
X < Y \land Y < Z & \rightarrow X < Z \quad \text{(transitivity)}
\end{align*}
\]

None of these integrity constraints is satisfied according to the theoremhood view (whether $T$ is formulated in if- or iff-form).\(^2\)

The same sentences can also be properties of the definition

\[
S_1 < S_2 \leftrightarrow \left[ S_1 = s_0 \land S_2 = \text{result}(A, S_3) \right] \lor \\
\left[ S_1 = \text{result}(A_1, S_3) \land S_2 = \text{result}(A_2, S_4) \land S_3 < S_4 \right]
\]

of the user-defined predicate $<$ in the situation calculus (see e.g. [KoSa94]), if the only intended model is the minimal Herbrand model (analogously to the case of the built-in predicate $>$).

Another view of integrity constraint satisfaction is the epistemic view proposed by Reiter [Re90]. Reiter argues that integrity constraints should be interpreted as statements about what a database knows (or should know). He formalizes this concept using a modal operator, $K$ for "known", to extend the syntax of first-order logic. The modal operator has to be inserted into integrity constraints at the proper places and different insertions may lead to different results. For example,

\[K\text{person}(X) \rightarrow K\text{male}(X) \lor K\text{female}(X)\]

means that every individual known to the database must either be known to

\(^2\)However, they can all be proved using an appropriate form of induction.
be male or known to be female (i.e. a gender has to be chosen), whereas

\[ K \text{person}(X) \rightarrow K[ \text{male}(X) \lor \text{female}(X) ] \]

is already satisfied if the database merely contains the disjunction \( \text{male}(p) \lor \text{female}(p) \) for every person \( p \) known to it (i.e. the choice of gender may be left open).

For certain choices of intended models, a careful rewriting of integrity constraints using the \( K \) operator may lead to a view of integrity constraint satisfaction similar to the propertyhood view. For example,

\[ Kp \rightarrow \text{false} \]

is satisfied, under the epistemic view, by the theory from example 2.4.1 (but the original integrity constraint, \( p \rightarrow \text{false} \), is not satisfied under the epistemic view), if the intended model is again the minimal Herbrand model.

The propertyhood view has the advantage that it does not require such rewriting and provides a greater flexibility in the choice of intended model (it appears unlikely that the epistemic view could simulate every kind of intended model).

An approach similar to the epistemic view is the metalevel view \([\text{Ko90, EsKo89}]\) which replaces the concept of the modal knowledge operator by the notion of provability, i.e. an integrity constraint

\[ Cond \rightarrow Conc \]

is interpreted as

"if \( Cond \) is provable, then \( Conc \) must be provable"

which can be formalized by means of the metalevel predicate \( \text{demo} \):

\[ \text{demo}(T, Cond) \rightarrow \text{demo}(T, Conc) \]

Similar comments as in the discussion of the epistemic view apply with respect to the comparison to the propertyhood view.

The view of integrity constraint satisfaction adopted in the abductive framework of Kakas and Mancarella \([\text{KaMa90}]\) also bears some resemblance to the propertyhood view. The "intended models" in this case are defined as those stable models (of the theory including the abducible part corresponding to

\[ 54 \]
in which the integrity constraints are true. These models are then called
*generalized stable models*, see also section 4.2.

Finally, Smolka [Sm91] defined *admissible guarded rules* for CLP which
can also be interpreted as integrity constraints true in the intended models,
which in his case are the minimal Herbrand models. While Smolka’s approach
is presented in the specialized context of CLP, it may be regarded as an in-
stance of the propertyhood view and is probably the conceptually closest to
the propertyhood view among the related approaches. It is further discussed
in section 4.3.4.

2.4.3 Integrity constraints for equality

The definition of equality in $T_0$ as an enumeration of all ground facts, i.e.
$t = t \leftrightarrow \text{true}$ for all ground terms $t$
does not entail (in the sense of the theoremhood view) many desirable proper-
ties of equality, in particular no negative consequences such as the disequality of
distinct ground terms. The axioms of Clark’s Equality Theory (*CET*, [Cl78])
are commonly used in LP to capture the intended meaning of equality. [Ho90]
lists them as follows:

(CET1) $t_1 \neq t_2$ for each pair $(t_1, t_2)$ of distinct constants

(CET2) $f_1(X_1, \ldots, X_m) \neq f_2(Y_1, \ldots, Y_n)$ for each pair of distinct function
symbols $f_1$ and $f_2$

(CET3) $f(X_1, \ldots, X_m) \neq t$ for each function symbol $f$ and each constant $t$

(CET4) $f(X_1, \ldots, X_m) \neq f(Y_1, \ldots, Y_m) \leftarrow X_1 \neq Y_1 \lor \ldots \lor X_m \neq Y_m$ for each
function symbol $f$

(CET5) $f(\ldots X \ldots) \neq X$ for each functional term $f(\ldots X \ldots)$ containing $X$

(CET6) $X = X$
(CET7) \( X = Y \leftrightarrow Y = X \)

(CET8) \( X = Z \leftrightarrow X = Y \land Y = Z \)

(CET9) \( f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_m) \leftrightarrow X_1 = Y_1 \land \ldots \land X_m = Y_m \) for each function symbol \( f \)

(CET10) \( p(X_1, \ldots, X_m) \leftrightarrow p(Y_1, \ldots, Y_m) \land X_1 = Y_1 \land \ldots \land X_m = Y_m \) for each predicate symbol \( p \)

**Theorem 2.4.1** (CET1)–(CET10) are true in the minimal Herbrand model of the definition of equality in \( \mathcal{T} \). Thus

\[ \mathcal{T}_b \models_{int} CET \]

**Proof:** (CET1)–(CET5) are disequalities between terms which do not match any equality fact in \( \mathcal{T}_b \). (CET6)–(CET8) are true in the minimal Herbrand model because equality is an equivalence relation. (CET9) and (CET10) are substitutions.

Since the intended interpretation of equality is fixed to that of the minimal Herbrand model, the CET axioms are properties of the definition of equality, i.e. \( \mathcal{T}_b \models_{int} CET \).

Theorem 2.4.1 is important for the soundness of the equality rewriting operation of the proof procedure, see definition 3.1.4. It follows that

\[ T \models_{int} \mathcal{IC} \cup CET \]

which suggests that it may be possible to regard the CET axioms as integrity constraints (possibly built-in),\(^3\) thus making the properties of equality explicit. Indeed, some of the CET axioms can easily be formulated as integrity constraints, e.g.

\(^3\)Integrity constraints for inequality and other built-in predicates could also be made available by means of system libraries. If deemed necessary for efficiency purposes, built-in ("black-box") constraint solvers could also be incorporated into the framework, just as in CLP. See also sections 4.3.2 and 4.3.5.
\[ f(X) = f(Y) \rightarrow X = Y \] (for every function symbol \( f \))

\[ f(X) = g(Y) \rightarrow \text{false} \] (for every pair of distinct function or constant symbols \( f \) and \( g \))

\[ p(X) \land X = Y \rightarrow p(Y) \] (for every predicate symbol \( p \))

(in each case all variables should be understood as vectors of zero or more variables).

Unfortunately, there are also a number of problems, some of more practical, some of more theoretical nature:

- When written as integrity constraints, the substitution axioms (CET10) cannot actually "perform" the substitution, they can only lead to the generation of new atoms, not to the disposal of the old ones (but see the deletion operation in chapter 5, although it is far from obvious how to solve this problem in practice even with deletion).

- The reflexivity axiom (CET6), which would have to be represented as \( \text{true} \rightarrow X = X \) when written as an integrity constraint, is hopelessly inefficient in practice.

- The symmetry axiom (CET7) is even worse, as it causes an infinite loop if not properly controlled.

- The "occur check" (CET5) requires a metalogical implementation.

For these reasons the \( CET \) axioms will not be supplied as integrity constraints for equality but will be implemented in the form of equality rewrite rules as part of the proof procedure (see definition 3.1.4). From the point of view of CLP, the equality rewrite rules may be regarded as a "transparent" constraint solver (cf. section 4.3.2). This would also be true for built-in integrity constraints representing the \( CET \) axioms, so there is little difference between the two approaches.
2.5 Semantics for normal LP

Conditions (A2) and (A3; ) of the declarative semantics defined in section 2.3 depend on which models are selected, by the user, as the intended models of \( \mathcal{T} \). There are numerous semantics proposed in the literature for normal logic programs; the most commonly used ones are presented in the surveys [ApBo94] and [Sh88]\(^4\) and reviewed here with respect to their suitability for the CALOG framework. This section may be skipped at first reading.

Recall from section 2.5 that intended models have to satisfy certain requirements. Those requirements, except for the second, are satisfied in all semantics to be considered here (attention will generally be restricted to Herbrand models). The second requirement, that \( \mathcal{T} \) has at least one intended model, may rule out certain choices of semantics for certain theories \( \mathcal{T} \).

Since definitions in \( \mathcal{T} \) are in if-and-only-if form, a completion semantics may seem a reasonable choice to provide a semantics for the CALOG framework. In the 2-valued completion semantics the intended models can be defined as the (Herbrand) models of the completion of the normal logic program \( \mathcal{P} \) corresponding to \( \mathcal{T} \). Recall that the completion of \( \mathcal{P} \) is the same as that of \( \mathcal{T} \) (see lemma 2.1.1).

**Example 2.5.1** The completion of the theory\(^5\)

\[ \mathcal{T}: \ p \leftrightarrow q \lor \neg r \]

is obtained by adding the two definitions

\[ q \leftarrow \text{false} \]
\[ r \leftarrow \text{false} \]

The completion of \( \mathcal{T} \) has only one model, in which \( p \) is true and \( q \) and \( r \) are

\(^4\)For ease of reference, theorems cited in this section are taken from the surveys rather than from the original sources (which are referenced in the surveys). The reader is also referred to the surveys for the precise definitions of some of the LP terminology used in this section.

\(^5\)For simplicity, the fact that the built-in theory \( \mathcal{T}_b \) is always part of \( \mathcal{T} \) will be ignored in this and the following examples. The external theory \( \mathcal{T}_e \) will be assumed to be empty, unless stated otherwise.
false.

So an initial goal $p$ has only one ideal answer, $\text{true}$, under the completion semantics. However, $q$ is an additional (non-ideal) good answer provided that there are no integrity constraints.

The main objections against the use of the 2-valued completion semantics in the LP literature are the possible inconsistency of the completion of a normal logic program (e.g. the completion of $p \leftarrow \neg p$ is inconsistent) and the general undecidability thereof (see theorem 18 of [Sh88]), although at least stratified and, more generally, call-consistent programs (i.e. programs where no predicate is defined, directly or indirectly, in terms of its own negation) always have a consistent completion (see theorem 4.6 of [ApBo94]; see [Sa90] for the even wider class of order-consistent programs which also have a consistent completion).

When completing the definitions of the built-in predicates in $\mathcal{T}$, the only intended model under the completion semantics is the minimal Herbrand model. In general, however, the completion semantics also allows non-minimal models, e.g.

$$\mathcal{T} : p \leftarrow p$$

has the two models $\emptyset$ and $\{p\}$. One could argue that the tautology $p \leftarrow p$ provides just as much information about $p$ as an empty theory and an empty theory would have only one intended model (the empty set). But even explicitly imposing minimality does not always yield satisfactory results with respect to tautologies as the following example shows:

Example 2.5.2 The theory

$$\mathcal{T} : p \leftarrow p$$

$$q \leftarrow \neg p$$

has two minimal models under the completion semantics: $\{p\}$ and $\{q\}$. Note that the program is stratified and both models are supported, yet the model

---

$^6$A model is supported if for every ground atom $A$ true in the model there is a ground
\{q\} may appear to be preferable to the model \{p\} as the only information given about \(p\) is a tautology.

The intended models of the standard model semantics for stratified programs, the perfect model semantics and the stable model semantics are all minimal (see theorems 6.7, 6.9, 6.20 of [ApBo94]) and coincide for stratified programs.\(^7\) For acyclic programs [ApBe90], which (like stratified programs) constitute a proper subclass of the locally stratified programs, even more semantics coincide (see theorem 11.1 of [ApBo94]), including the 2-valued completion semantics discussed above (and also the other semantics discussed below). These results imply that for large classes of theories there are more or less canonical choices for the notion of intended model.

**Example 2.5.3** The theory in example 2.5.2 is not acyclic as

\[ p \leftarrow -p \]

constitutes a cycle. Example 2.5.2 was given to illustrate a weakness of the 2-valued completion semantics, and indeed the other semantics (which coincide in the example since it is stratified) all yield the more intuitive result that there is only one intended model in which \(p\) is false and \(q\) is true.

In order to give a meaning to normal logic programs with "bad" definitions like \(p \leftarrow \neg p\), some semantics resort to three-valued interpretations in which the truth value of the problematic atoms is left undefined. Three-valued interpretations could also be applied to theories in the CALOG framework. One example is the well-founded semantics. Every normal logic program \(P\) has a unique well-founded model, and this model yields a three-valued model of \(\text{Comp}(P)\) (theorem 7.7 of [ApBo94]). This implies that the well-founded semantics is also compatible with the CALOG framework's semantics. Moreover, a total well-founded model (i.e. one which determines the truth value of instance of a definition in \(T\) with head \(A\) and a body which is true in the model (adapted from [Sh88]).

\(^7\)For locally stratified programs the perfect and stable model semantics still coincide and there is a unique perfect and stable model (theorem 6.20 of [ApBo94]).
every atom and thus is a two-valued model) is a stable model, and in this case the stable model is unique (corollary 7.8 of [ApBo94]). For locally stratified programs, this is always the case, so the well-founded semantics also coincides with the perfect model semantics in this case.

Example 2.5.4 (Two-loop and three-valued models)

\[ p \leftrightarrow \neg q \]
\[ q \leftrightarrow \neg p \]

The two-loop theory has two intended models under the completion and stable model semantics, one in which \( p \) is true and \( q \) is false and one in which \( q \) is true and \( p \) is false. Neither \( p \) nor \( q \) nor their negations are properties of the theory under those semantics. If the definition

\[ r \leftrightarrow p \lor q \]

is added, then \( r \) is true in both of the two intended models and is a property of the theory. Hence an initial goal \( r \) has \text{true} as an ideal answer. But any attempt to reduce \( r \) to \text{true} (i.e. by the proof procedure) is doomed to go into an infinite loop. The proof procedure to be defined in the next chapter is indeed incomplete for the two-valued completion semantics and the stable model semantics.

A way to rectify the incompleteness problem is to use the well-founded semantics. In three-valued logic \( p \) and \( q \) are both assigned the truth value \text{undefined}. Then their disjunction also has the truth value \text{undefined} and \( r \) is no longer true in the (unique) three-valued intended model.

Another semantics, which also avoids this incompleteness problem and which is compatible with the CAILOG framework’s notion of intended model, is the three-valued completion semantics of [Ku87, DeDS93] which has been used in SLDNFA [DeDS92, DeDS97] and the Iff Proof Procedure [Fu96, FuKo96]. It will be discussed in more detail in the comparison with these two proof procedures in section 4.2.2.2. The difference between the three-valued completion semantics and the well-founded semantics may be best illustrated by the fact
that for the theory

\[ p \leftrightarrow p \]

the truth value of \( p \) is false in the unique model of the well-founded semantics, whereas the three-valued completion semantics also has a model in which \( p \) is undefined (the truth value of undefined \( \leftrightarrow \) undefined being true in the underlying three-valued logic used in [Ku87]). This implies that the three-valued completion semantics is the weakest of those discussed here and the proof procedure defined in the next chapter is "most complete" with respect to the three-valued completion semantics.
Chapter 3

Proof Procedure

This chapter formalizes the CALOG framework's proof procedure, called the CALOG proof procedure (CPP). First the operations of the CPP are defined, then formal results with respect to the declarative semantics defined in the previous chapter are presented.

The CPP is based on, but generalizes and simplifies, two abductive proof procedures, the Iff Proof Procedure [Fu96, FuKo96] and SLDNFA [DeDS92, DeDS97]. The CPP is more general as it is not geared towards abduction and as it has to provide for several new concepts which the frameworks underlying the other proof procedures did not feature, e.g. suspension, the liberty of the user choosing the intended models and the propertyhood view of integrity constraint satisfaction. A detailed comparison between the CPP and these two proof procedures is conducted in the section on ALP in chapter 4.

The version of the CPP presented in this chapter may serve as a kernel which can be extended and instantiated to yield numerous more practical and more efficient proof procedures for different purposes. Possible extensions are discussed in the comparison with the two related proof procedures and in chapter 5. In practice, the two most important extensions to obtain good computational results using the implementations (see chapter 6) are the CPD techniques discussed in section 5.1 and the deletion operation discussed in section 5.4.
3.1 The CALOG Proof Procedure (CPP)

3.1.1 Operations of the proof procedure

Definition 3.1.1 Let \( G_0 \) be an initial goal. A derivation of \( G_n \) from \( G_0 \) is a finite sequence \( G_0, G_1, \ldots, G_n \) where \( G_1 \) is obtained from \( G_0 \) by conjoining all integrity constraints in \( IC \) to \( G_0 \), and, for \( 1 \leq i \leq n \), \( G_{i+1} \) is obtained from \( G_i \) by applying one of the following operations:

- **unfolding** a reducible atom, using a definition in \( T' \)
- **logical equivalence transformation**
- **propagation**, which is a form of resolution between subgoals
- **equality rewriting**

The four operations are defined in the following.

Definition 3.1.2 (Unfolding)

Let \( A \) be a reducible atomic subgoal or a reducible atom in the condition of an implication in \( G_i \). If there is a definition \( H \leftrightarrow D_1 \lor \ldots \lor D_m \) in \( T' \) and a substitution \( \sigma \) such that \( A = H \sigma \), then replace \( A \) by \( (D_1 \lor \ldots \lor D_m) \sigma \).

Recall that, for a reducible atom \( A \), there is one and only one definition such that \( A \) is an instance of the head of the definition, so the unfolding operation itself is deterministic. Moreover, atoms which do not unify with the head of any definition are not unfolded to \( \text{false} \) even if there are definitions for the atom's predicate (e.g. \( 1<0 \) is not unfolded to \( \text{false} \), cf. the examples given after definition 2.2.2). If such atoms were unfolded to \( \text{false} \), then goals having good answers might be reduced to \( \text{false} \) (such as \( 1<0 \) which is a good answer to itself provided there are no integrity constraints), which may seem desirable in some cases, but undesirable in others (e.g. partially defined predicates in abduction). By not unfolding these atoms to \( \text{false} \) the choice is left to the user who can
specify appropriate integrity constraints to explicitly reject unwanted answers (e.g. anti-symmetry $X < Y \land Y < X \rightarrow \text{false}$ eliminates $1 < 0$ and $\text{plus}(X, Y, Z) \land \text{plus}(X, Y, Z') \rightarrow Z = Z'$ eliminates $\text{plus}(2, 2, 5)$).

Definition 3.1.3 (Logical equivalence transformation)

A logical equivalence transformation is the application of one of the following equivalences as a rewrite rule to replace a formula which matches the left-hand side by the corresponding formula on the right-hand side. The formula to be replaced may occur anywhere in the goal except in the conclusion of an implication.

- $(A \lor B) \land C \leftrightarrow (A \land C) \lor (B \land C)$ (splitting)
- $(A \lor B) \rightarrow C \leftrightarrow (A \rightarrow C) \land (B \rightarrow C)$ (normalization of implications)
- $A \land \text{true} \leftrightarrow A, A \land \text{false} \leftrightarrow \text{false}$
- $A \lor \text{true} \leftrightarrow \text{true}, A \lor \text{false} \leftrightarrow A$
- $(\text{true} \rightarrow A) \leftrightarrow A, (\text{false} \rightarrow A) \leftrightarrow \text{true}$
- $\neg A \leftrightarrow (A \rightarrow \text{false})$, if $\neg A$ is a goal literal
- $(\neg A \land B \rightarrow C) \leftrightarrow (B \rightarrow A \lor C)$

The last two equivalence transformations deal with negative literals, introduced either in the initial goal or by unfolding (remember that integrity constraints have only positive literals in both conditions and conclusions). As their exhaustive application completely eliminates negative literals from a goal, no other operations of the proof procedure are required to deal with negative literals.

Equality atoms are introduced into goals both by unfolding and by propagation (see below). Equality rewriting deals with such equalities, and it is defined via the following rewrite rules, taken from [Fu96] where they have been adapted from [MaMo82].
Definition 3.1.4 (Equality rewriting)

*Equality rewriting* is the application of one of the equivalences (EQ1)-(EQ5) as rewrite rules or of the term substitution rule (EQ6). Rules (EQ1)-(EQ5) are applied to atomic subgoals or to atoms in the conditions of implications in a goal.¹

(EQ1) \[ f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \iff t_1 = s_1 \land \ldots \land t_n = s_n \quad (n \geq 1, f \text{ any function symbol}) \]

(EQ2) \[ f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m) \iff \text{false} \quad (n, m \geq 0; f, g \text{ any distinct function or constant symbols}) \]

(EQ3) \[ x = x \iff \text{true} \]

(EQ4) \[ y = x \iff x = y, \text{ if either } x \text{ is a variable and } y \text{ is not or both } x \text{ and } y \]

are variables, but \( x \) is universally quantified and \( y \) is not

(EQ5) \[ x = t \iff \text{false} \quad \text{where the term } t \text{ contains } x \quad \text{(occur check)} \]

(EQ6) If \( x \) is any variable and \( x = t \) is an atomic subgoal of a goal \( G \) and none of (EQ1)-(EQ5) can be applied to \( x = t \), then replace all other occurrences of the same variable \( x \) in \( G \) by \( t \) (not including \( x = t \)).²

If \( x \) is a universally quantified variable and \( x = t \) occurs in the condition of an implication and (EQ1)-(EQ5) cannot be applied to \( x = t \), then replace all occurrences of the same variable \( x \) within the implication by \( t \).³

¹Just like unfolding and logical equivalence transformation, the rules (EQ1)-(EQ5) are not applied to atoms in the conclusion of an implication. This is based on the idea that conclusions of implications have to be considered only after the conditions have been completely resolved away (by propagation).

²The equality \( x = t \) itself could also be replaced and then rewritten to \text{true} by (EQ3) if \( x \) is an existentially quantified variable. Otherwise, if \( x \) is a free variable, the assignment has to be kept for the sake of condition (A2) of the answer definition.

³Including \( x = t \) to \( t = t \), which is then reduced to \text{true} by (EQ3).
(EQ4) implements the symmetry axiom of CET in such a way that rewriting always terminates. The substitution rule (EQ6) implements transitivity and the two substitution axioms of CET. Example 3.1.1 below illustrates some of the rewrite rules.

The rewrite rules (EQ1)-(EQ5) and the substitution rule (EQ6) constitute a sound and complete implementation of the CET axioms (CET1)-(CET10) specified in section 2.4.3 (cf. [Fu96] and [MaMo82]). It has already been discussed in section 2.4.3 that it might be desirable, from a theoretical point of view, to regard the CET axioms, which are properties of the built-in definition of equality, as integrity constraints for equality. The equality rewriting operation as defined above may now be seen as an optimized form of propagation with such integrity constraints where, among other practical improvements, the redundant half of the equivalence is deleted after propagation.

From the CLP point of view, the rewrite rules implement a transparent (or glass-box) constraint solver for equality over the Herbrand domain, cf. section 4.3.2.

The propagation operation is easiest to define for goals which are in the form of a conjunction of atoms and (normalized) implications. Any goal can be transformed into a disjunction of such goals by logical equivalence transformations. A more efficient form of propagation, which is similar to non-clausal resolution [Mu82] and which also applies to goals not in this form, is discussed in chapter 5.

**Definition 3.1.5 (Propagation)**

*Propagation* adds to a conjunction, which contains the two subgoals

\[ p(s_1, \ldots, s_n) \land Cond \rightarrow Conc \]

and

\[ Susp \rightarrow p(t_1, \ldots, t_n) \lor Conc', \]

the resolvent

\[ Cond \land Susp \land s_1 = t_1 \land \ldots \land s_n = t_n \rightarrow Conc \lor Conc'. \]

Here, *Cond* and *Susp* are conjunctions of zero or more atoms, *Conc* is a
disjunction of one or more atoms and \( Conc' \) is a disjunction of zero or more atoms.\(^4\)

Since the equality rewriting operation deals with equality atoms, there is no need to integrate unification into the propagation operation. Note that this would be a non-trivial task due to the different quantifications of variables; the definition of an operation similar to propagation in SLDNFA suffers from these complications (cf. section 4.2.2.3).

Intuitively, \( Susp \) consists of suspended atoms which cannot be unfolded before applying propagation. Restricting propagation to the case in which \( Susp \) is empty corresponds to \( P_1 \)-resolution and requiring all condition atoms \( (p(s_1, \ldots, s_n) \) and the atoms in \( Cond \) to be resolved away at once is hyper-resolution. These and other possible restrictions of propagation are discussed in chapter 5.

Note how an expression \( A \land \neg A \) can be reduced to \( false \) by logical equivalence transformations and propagation: first \( \neg A \) is rewritten to \( A \rightarrow false \), then propagation can be applied and adds \( false \) to the conjunction. After another logical equivalence transformation, the conjunction collapses to \( false \).

Example 3.1.1

\( T_a : \text{empty} \)

\( IC : p(X) \land q(Y) \land X>Y \rightarrow false \)

\( X>Y \land Y>Z \rightarrow X>Z \)

\( G_0 : p(A) \land q(0) \land A>1 \)

\( G_1 \) is \( G_0 \) with the two integrity constraints conjoined. Propagation, using the first integrity constraint with \( p(A) \), generates

\( q(Y) \land X>Y \land X=A \rightarrow false \)

which is transformed into

\(^4\)If \( Susp \) contains no atoms, then neither does \( Conc' \) (because splitting would have been applied) and the expression \( Susp \rightarrow p(t_1, \ldots, t_n) \lor Conc' \) is to be understood as the atom \( p(t_1, \ldots, t_n) \).
\(q(Y) \land A\succ Y \rightarrow \text{false}\)

by equality rewrite rules (EQ6) and (EQ3). The resolvent can then be used with \(q(0)\) to propagate again, resulting in (after equality rewriting using rules (EQ6) and (EQ3) again)

\[A\succ 0 \rightarrow \text{false}\]

Notice that \(A\succ 0\) is suspended. Using the transitivity rule with \(A\succ 1\) from the initial goal yields (again after equality rewriting)

\[1\succ Z \rightarrow A\succ Z\]

Notice that \(1\succ Z\) is suspended. Up to this point propagation has only been used in the special case where \(\text{Susp}\) in the definition of propagation is empty. Now, however, the two last implications can be used to propagate in the more general way to yield

\[1\succ Z \land A=A \land 0=Z \rightarrow \text{false}\]

which, using rewrite rules (EQ3) for \(A=A\), (EQ4) for \(0=Z\), then (EQ6) to replace the \(Z\) in \(1\succ Z\) and (EQ3) to remove \(0=0\), can be rewritten to

\[1\succ 0 \rightarrow \text{false}\]

Thus a reducible atom, \(1\succ 0\), appears as the condition of an implication which was obtained from two implications which only contained suspended atoms. \(1\succ 0\) can be unfolded to \(\text{true}\), and a logical equivalence transformation adds \text{false} to the goal (which then collapses to \text{false}).

The example has shown that it may be necessary to propagate between non-degenerate implications (where the \(\text{Cond}\) and \(\text{Susp}\) parts are not trivial). This will again be discussed in section 5.2.

The definition of propagation is rather inefficient and prevents the proof procedure from terminating in many cases in which termination should be expected. In example 3.1.1, changing the initial goal to

\[G'_6: p(A) \land q(2) \land A\succ 1\]

gives a goal which is consistent with the definition of \(\succ\) and the integrity constraints (i.e. it is a good answer), but propagation does not terminate because infinitely many resolvents can be recursively obtained by propagation with
transitivity; cf. example 3.4.1 in chapter 5 where this problem is addressed and where possible solutions are discussed.

Section 3.3 presents formal soundness results relating the proof procedure to the declarative semantics defined in the previous chapter. Roughly speaking, the CPP operations unfolding and logical equivalence transformation reduce the initial goal to the form $D \lor Rest$ so that a computed answer $A$ extracted from the goal disjunct $D$ (the extraction of computed answers is defined in the next section) satisfies conditions (A1) and (A2) of definition 2.3.2. Propagation and equality rewriting ensure that $A$ is consistent with $TCUCET$ and therefore satisfies at least condition $(A_{3w})$ of the weak answer definition. Unfolding atoms in conditions of integrity constraints is needed to verify condition $(A_{3g})$ of the good answer definition, but suspension may prevent this.

3.1.2 Extracting computed answers

The aim of a derivation starting from an initial goal $G_0$ is to generate one or more computed answers to $G_0$. Every computed answer to $G_0$ has to satisfy conditions (A1) and (A2) and at least $(A_{3w})$ of definition 2.3.2.

Suppose that there is a derivation of the goal $G_n = D \lor Rest$ from the initial goal $G_0$ and that the operations of the proof procedure have been exhaustively applied to $D$.

One possibility to extract a computed answer from $D$ is to simply take $D$ itself (possibly with some modifications so that it satisfies condition (A1)). But part of the information in $D$ may either be irrelevant with respect to the initial goal $G_0$ or hard to interpret. There are different ways of extracting more meaningful and more intelligible computed answers and several of them have been proposed and discussed in the work on SLDNFA and the Iff Proof Procedure. The approach taken here is most closely related to, but slightly simplifies the one taken in [DeDS97] (cf. section 4.2.2.4).

The following lemma shows which information could be removed from $D$. 

70
Lemma 3.1.1 Let $C$ be a conjunct of $D$, i.e. $D = C \land D'$. If $C$ is

- an integrity constraint
- a resolvent obtained from a propagation operation
- the result of applying one or more CPP operations other than propagation to either an integrity constraint or a resolvent obtained from a propagation operation

then

$\mathcal{T} \models_{\text{int}} \mathcal{V}[D \leftrightarrow D']$

Proof: If $C$ is an integrity constraint, then the lemma holds because $\mathcal{T} \models_{\text{int}} IC$. Unfolding, logical equivalence transformations and equality rewriting all preserve equivalence within the intended models of $\mathcal{T}$ (see lemma 3.3.1), so if $C$ is derived from an integrity constraint in one or more steps of these operations, then the lemma also holds.

If $C$ is obtained as the resolvent of a propagation operation, then it is logically redundant, i.e.

$\models \mathcal{V}[D \leftrightarrow D']$

Again, unfolding, logical equivalence transformations and equality rewriting preserve equivalence within the intended models of $\mathcal{T}$, so the lemma continues to hold if $C$ has been derived in one or more of these ways from a resolvent of a propagation operation.

So if $D$ satisfies condition (A2) of the semantic answer definition, then so does $D'$ where $D'$ is obtained from $D$ by removing one or several (or all) of the conjuncts of $D$ for which the lemma holds.

The question remains whether all information that could be removed from $D$ should indeed be removed. The following example shows that some pieces of the "redundant" information may be of interest while others may be irrelevant (formalization follows in definition 3.1.6).
Example 3.1.2

$\mathcal{T}_0$: empty

$\mathcal{IC}: a \rightarrow b$

$G_0: a$

The derivation terminates with

$G_2 = D = a \land b \land (a \rightarrow b)$

According to lemma 3.1.1, both the integrity constraint and the atom $b$ could be removed from $D$ without affecting condition (A2). This would undo the propagation step and leave just $a$, i.e. the initial goal. But whereas the integrity constraint is irrelevant with respect to $G_0$ in the sense that it will become part of any $D$ in any derivation starting from any initial goal, the atom $b$ is generated only if $G_0$ contains $a$, so one might prefer to keep $b$ and return

$a \land b$

as the computed answer.

Now suppose that the initial goal is the same but that there are two additional integrity constraints:

$a \land c \rightarrow d$

e $\rightarrow a$

Now two more steps of propagation are possible, one between the first of these integrity constraints and the atom $a$, the other among the two integrity constraints. The two resolvents are:

$c \rightarrow d$

$c \land e \rightarrow d$

Again, according to lemma 3.1.1, both of these resolvents could be removed from $D$. But whereas the second resolvent is generated independently of what $G_0$ is, the first resolvent depends on $G_0$ and one might prefer to keep it in (or at least use it for) the computed answer extracted from $D$.

The example suggests that the proof procedure should keep track of which

5Moreover, the first resolvent happens to subsume the second. Subsumption could generally be applied to simplify goals at any stage in a derivation, cf. section 5.3.
information has been derived exclusively by means of integrity constraints.

**Definition 3.1.6** An implication in $D$ is called *irrelevant* (for the extraction of computed answers) if it is

- an integrity constraint

- a resolvent of two irrelevant implications (obtained by propagation)

- the result of applying one or more CPP operations other than propagation to an irrelevant implication

All other implications in $D$ are called *relevant*.

Note that lemma 3.1.1 holds for any irrelevant implication. So the idea is to eliminate exactly the irrelevant implications from $D$ when extracting computed answers. ⁶

After this has been done, $D$ may still not be in the right form to satisfy condition (A1). Condition (A1) requires all atoms occurring in $D$, whether as atomic subgoals or as condition atoms in implications, to be suspended. But unfolding is not applied to the conclusion atoms of implications, so those atoms might be reducible and they might be equivalent to false. In the propositional case, there is a simple solution to this problem: replace the conclusions of all relevant implications by false when extracting computed answers. To see why this makes sense and to motivate a solution for the general case, consider the following example.

**Example 3.1.3** Suppose that $D$ is

$$p(f(A)) \land [ p(f(X)) \rightarrow r(X) ] \land r(A)$$

where $A$ is a free variable from the initial goal and $X$ is universally quantified over the implication.

⁶The reader may wonder why definition 3.1.6 is restricted to implications. Atoms could be obtained in the same way as irrelevant implications, but are not called irrelevant. This is to allow computations based entirely on integrity constraints generating (or discovering), in a bottom-up manner, the definitions of external predicates. See section 6.5.2 for an example.
Clearly, the atoms \( p(f(A)) \) and \( r(A) \) have to become part of a computed answer extracted from \( D \). To ensure that the implication in \( D \) is true regardless of what the definition of \( r \) is (note that \( r \) might be an external predicate so that its definition is inaccessible), \( p(f(X)) \) should not hold for any \( X \neq A \). No further assumptions have to be made with respect to the truth value of instances of \( p(t) \) where \( t \) does not unify with \( f(X) \) (note that it does not matter whether \( p \) is user-defined, built-in or external — since the operations of the proof procedure have been exhaustively applied, \( p(f(X)) \) is suspended in any case) and neither is any assumption needed with respect to the definition of \( r \). So a reasonable choice for a computed answer extracted from \( D \) is

\[
p(f(A)) \land r(A) \land [ p(f(X)) \land X \neq A \rightarrow \text{false} ]
\]

Note that the implication could be equivalently written as \( p(f(X)) \rightarrow X = A \).

This choice is reasonable in the sense that the computed answer is guaranteed to satisfy condition (A2) without having to know what the definitions of \( p \) and \( r \) are.

**Definition 3.1.7** (Computed answer extraction)

Suppose there is a derivation of the goal \( G_n \) from the initial goal \( G_0 \) and that \( D \neq \text{false} \) is a disjunct of \( G_n \) to which the operations of the proof procedure have been exhaustively applied.

For every implication

\[ \text{Cond} \rightarrow \text{Conc} \]

in \( D \), which is not irrelevant in the sense of definition 3.1.6, construct an implication

\[ \text{Cond} \land \text{DisEq} \rightarrow \text{false} \]

with \( \text{DisEq} \) being the conjunction of all disequalities obtained in the following way: for every atom \( p(X) \) in \( \text{Cond} \) (\( X \) being an \( n \)-tuple of terms, \( n \geq 0 \)) add the disequalities \( X \neq t_1, \ldots, X \neq t_n \) to \( \text{DisEq} \) where \( p(t_1), \ldots, p(t_n) \) are all the atoms of the predicate \( p \) occurring in \( D \). Variable quantifications remain unchanged.
The conjunction of all implications thus obtained and of the atoms occurring in $D$ is the *computed answer* to $G_0$ extracted from $D$.

Note that the transformation of implications into denials when extracting computed answers may exclude some semantic answers from being computed. For instance, in example 3.1.3, any semantic answer in which two different instances of $p(f(X))$ are true is not computed, although there might be such answers, even ideal ones. But the correctness of such answers cannot be verified (due to suspension), and soundness of the proof procedure appears to be more important than completeness (which, for other reasons, cannot be shown in general anyway).

Ignoring the irrelevant implications in $D$ not only simplifies a computed answer, but is necessary to avoid generating denials (as part of the answer) which are inconsistent with $T$. For example, suppose the transitivity rule for $>$ is part of the integrity constraints:

$$X > Y \land Y > Z \rightarrow X > Z$$

For simplicity, suppose there are no $>$-atoms in $D$. Then transforming the integrity constraint into a denial in the way defined by definition 3.1.7, yields

$$X > Y \land Y > Z \rightarrow \text{false}$$

which is absurd, i.e. inconsistent with the definition of $>$. Thus any computed answer containing this denial would not be a good answer.

If the definitions in $T$ and the initial goal do not contain negative literals, then any implication present in $D$ must have been derived from an integrity constraint or the resolvent of a propagation operation. Thus all implications in $D$ are redundant in the sense of lemma 3.1.1 (but they are not necessarily all irrelevant in the sense of definition 3.1.6) and could be removed before extracting computed answers. This allows a much simpler computed answer definition: just take the conjunction of all atoms in $D$. 

75
3.1.3 Safety of operations

Proof procedures based, in some way or other, on SLDNF (see sections 3.4 and 4.2.2) have to address and either avoid or solve the problem of constructive negation. The problem can be illustrated by the logic program consisting of the single clause

\[ p(X) \leftarrow X=a \]

and the goal

\[ \neg p(X) \]

The intended logical reading of this goal in standard LP is \( \exists X \, \neg p(X) \) and, assuming the domain consists of more elements than just \( a \), this goal should succeed. But if executed in PROLOG, the goal will fail because PROLOG tries \( p(X) \), which succeeds (with \( X=a \)) and concludes that \( \neg p(X) \) must thus fail — in other words (as is well-known), PROLOG's implementation of negation as failure is unsound (with respect to the interpretation of the goal as \( \exists X \, \neg p(X) \) ).

One way to restore soundness in this case is to prevent the selection of non-ground negative literals. A goal which consists only of such literals cannot be processed further and the computation is said to flounder (i.e. it stops without returning answers). Thus soundness is restored, but completeness is sacrificed.

The problem of constructive negation is to find a sound procedure which does not flounder in situations like this. One solution is described in [Ch88] and is related to the approach taken here (and also in [Fu96] and [DeDS97]). The main idea of [Ch88] is to represent substitutions explicitly in the form of equality atoms so that, if \( A_1, \ldots, A_n \) are all the (semantic) answers to an initial goal \( G_0 \), then

\[ \forall [G_0 \leftrightarrow A_1 \lor \ldots \lor A_n] \]

holds in the Clark completion of the underlying theory (or logic program) and thus

\[ \forall [\neg G_0 \leftrightarrow \neg (A_1 \lor \ldots \lor A_n)] \]

also holds in the completion. For example, if the program consists of the clause
\[ p(X) \leftarrow X = a \]
then the goal
\[ p(X) \]
is equivalent to \( X = a \) in the completion
\[ p(X) \leftarrow X = a \]
of the program, and the goal
\[ \neg p(X) \]
is equivalent to \( X \neq a \). An equivalent result is obtained by the CPP: first it transforms the negative goal literal \( \neg p(X) \) into
\[ p(X) \rightarrow false \]
Now the condition of the implication can be unfolded, and the computation terminates with
\[ X = a \rightarrow false \]
which is also the computed answer and is equivalent to the disequality \( X \neq a \)
(below the two representations of disequalities will be regarded as interchangeable).

So the CPP solves the problem of constructive negation in some cases. However, as illustrated in the following, the CPP cannot deal correctly with universally quantified variables occurring in atomic subgoals. Since the initial goal contains only free variables and all variables in the body of a definition which do not occur in the head are existentially quantified, such atomic subgoals can only be introduced into the goal by a logical equivalence transformation which rewrites a universally quantified implication
\[ \forall X \left[ true \rightarrow p(X) \right] \]
into the universally quantified atom \( \forall X \ p(X) \).

The implication \( true \rightarrow p(X) \) may occur either directly as an integrity constraint or may have been derived, in one or more steps, from another implication (which is either an integrity constraint or has been obtained by propagation or by rewriting negative literals using logical equivalence transformations), e.g.
q(X) → p(X)
yields the above implication after unfolding, if there is a definition
q(X) ↔ true

There are two major problems which may result from atomic subgoals with universally quantified variables. The first is a problem with the unfolding operation. If

∀X p(X)

occurs in the goal and the definition for p(X) is

p(X) ↔ q(X, Y)

then the correct reading of the result of unfolding would be

∀X ∃Y q(X, Y)

Thus quantifiers may become arbitrarily nested, making goals hard to interpret and also causing problems when extracting computed answers.

The second problem is even more severe. A disjunction

∀X ( p(X) ∨ q(X) )

which may have been introduced by either propagation or unfolding, is not equivalent to

( ∀X p(X) ) ∨ ( ∀X q(X) )

and thus cannot be processed correctly by the splitting operation.

One way to avoid these problems is to never introduce atoms with universally quantified variables into the goal. As has already been said, there is only one way such atoms may be introduced into the goal, so a simple safety restriction suffices:

Definition 3.1.8 (Safety of operations)
The logical equivalence transformation

(true → A) ↔ A

is called unsafe if A contains a universally quantified variable. All other applications of this transformation and all other operations of the proof procedure are called safe.

78
Definition 3.1.8 considerably simplifies the safety criterion used by SLDNFA (see also section 4.2.2.1). This is mainly due to the fact that SLDNFA does not have explicit integrity constraints.

Lemma 3.1.2 Atomic subgoals with universally quantified variables never appear in derivations in which only safe operations are applied starting from the initial goal.

Proof: The initial goal does not contain atoms with universally quantified variables. All variables in the body of a definition which do not appear in the head are existentially quantified, so unfolding cannot introduce atoms with universally quantified variables. Equality rewriting does not transform implications into atoms. Propagation generates implications, not atoms.

The only logical equivalence transformation which rewrites implications into atoms is

\[(\text{true} \to A) \leftrightarrow A\]

and this operation is only applied if it is safe to do so, i.e. if \(A\) does not contain universally quantified variables.

The result follows by induction on the length of the derivation. 

A way to ensure that operations are always safe is to require both definitions and integrity constraints to be range-restricted in the following sense:

Definition 3.1.9 (Range-restrictedness)
A definition or implication or atom is called range-restricted if

- every variable in the head of the definition appears in a positive literal in each disjunct of the body of the definition

- every variable in a negative literal in a disjunct of the body of the definition appears in a positive literal in the same disjunct of the definition
• every universally quantified variable in the conclusion of the implication
  appears in a positive literal in the condition of the implication

• the atom contains no universally quantified variables

In the first three cases the positive literal must not be an equality \( t = s \)
where both \( t \) and \( s \) contain universally quantified variables.

The last requirement of the definition is needed because an equality between
variables may be transformed into \( \text{true} \) by equality rewrite rules (EQ6) and
(EQ3), e.g. the definition
\[
p(X, Y) \leftrightarrow X = Y
\]
is equivalent to
\[
p(X, X) \leftrightarrow \text{true}
\]
which is not range-restricted.

Lemma 3.1.3 If all definitions and integrity constraints are range-restricted,
then all operations applied during a derivation from any initial goal are safe.

Proof: It suffices to show that all atoms and implications occurring in the
derivation are range-restricted. Then an implication true \( \rightarrow A \), where \( A \)
contains a universally quantified variable, can never occur because it is not
range-restricted.

The initial goal contains no implications, and all atoms in the initial goal
are range-restricted because they can contain only free variables.

Since integrity constraints are range-restricted, no non-range-restricted im-
plications are introduced into the goal by adding the integrity constraints to
the initial goal in the first step of the derivation.

Since definitions are range-restricted, unfolding preserves range-restricted-
ness of implications. Furthermore, it is well known that the resolvent of two
range-restricted clauses (in this case either two implications or one implica-
tion and one atom) is again range-restricted, so propagation never generates
implications which are not range-restricted.
Due to the special restriction on equality in the definition of range-restrictedness, equality rewriting cannot introduce a non-range-restricted atom or implication either. Checking that all logical equivalence transformations never introduce such atoms or implications is rather tedious and left as an exercise. □

Fung’s Iff Proof Procedure [Fu96] imposes the range-restrictedness requirement on definitions and integrity constraints to guarantee the safety of operations. This unnecessarily restricts the expressive power of the framework. Note especially that non-range-restricted integrity constraints and implications, including those of the type \( \text{true} \rightarrow A \), which occur in a CPP derivation, can be used for propagation. For example,

\[
\text{true} \rightarrow p(X) \quad \text{and} \quad p(a) \rightarrow \text{false}
\]

can be used to generate \text{false}, even though it is not possible to transform the first implication into the (universally quantified) atom \( p(X) \).

### 3.2 Order of the operations

In the simple case where integrity constraints contain only external predicates (as in the approach to SQO discussed in section 4.1), the derivation of computed answers could be divided into two phases. In the first phase unfolding and splitting are used to satisfy conditions (A1) and (A2); in the second phase propagation and equality rewriting (together with logical equivalence transformations) are used to satisfy at least condition (A3\(_{\omega} \)), ideally condition (A3\(_{\ell} \)). Both to deal with the more general case and to improve efficiency by detecting violations of condition (A3\(_{\omega} \)) earlier, the operations should be interleaved.

One straightforward strategy is to always apply logical equivalence transformations (except splitting) first, equality rewriting second and then exhaustively apply propagation before doing one step of unfolding. This sequence of operations is repeated until no further steps of unfolding are possible.
Then splitting a disjunction (if any) is the only remaining option, and the procedure works on both disjuncts in the same way (if possible in parallel; otherwise a sequential search method has to be identified, the simplest being depth-first).

The described strategy has been adopted in the implementation, see chapter 6.

Splitting should be delayed as long as possible because the same operations may have to be applied redundantly to both disjuncts of a disjunction. Propagation (together with the other operations) can generate false and make splitting superfluous.

The results reported in the next sections do not depend on a particular ordering of the operations. The issue of confluence has not been investigated; see [AbFrMe96] and [Ab97] for a discussion of potential problems and solutions in the related framework of CHRs (cf. section 4.3.5).

### 3.3 Soundness results

If only unsafe operations are applicable at any point in a derivation, the proof procedure must flounder and not return any computed answers. If it did attempt to extract computed answers from a floundered computation, these might be incorrect, e.g. if

\[
\text{true} \rightarrow p(X)
\]

is part of the goal (with X being universally quantified) and is not irrelevant in the sense of definition 3.1.6, then

\[
\text{true} \rightarrow \text{false}
\]

becomes part of the computed answer, which is therefore equivalent to false, although the disjunct of the goal from which it was extracted need not be equivalent to false. Consequently, the proof procedure would not be refutation sound (see below).

Therefore, soundness results can be obtained only for derivations in which
all operations are safe. Such derivations will be called \textit{safe derivations}.

3.3.1 Soundness of the operations

The operations of the CPP are all sound in the following sense:

Lemma 3.3.1 Let $G_i, i \geq 1$, be a goal in a safe derivation starting from $G_0$ and let $G_{i+1}$ be the goal obtained from $G_i$ by applying a single operation of the CPP. Then

$$ T' \cup CET \models \forall[G_i \leftrightarrow G_{i+1}] $$

Proof: If the operation used to derive $G_{i+1}$ from $G_i$ is unfolding, then

$$ T' \models G_i \leftrightarrow G_{i+1} $$

because an atom in $G_i$ is replaced by an atom equivalent in $T'$. If the operation is propagation, then

$$ \models G_i \leftrightarrow G_{i+1} $$

because resolution is a sound logical inference. If the operation is equality rewriting, then

$$ CET \models G_i \leftrightarrow G_{i+1} $$

because equality rewriting is a sound implementation of the CET axioms.

If the operation is a logical equivalence transformation, the two goals are unconditionally equivalent (in the case of splitting and normalization of implications due to the safety assumption, otherwise trivially).

The first step in a derivation simply adds the integrity constraints to the initial goal. Thus, we get from lemma 3.3.1 by induction:

Corollary 3.3.1 If there is a safe derivation of $G_n$ from $G_0$, then

$$ T' \cup IC \cup CET \models_{\inf} \forall[G_0 \leftrightarrow G_n] $$

Since $T \models_{\inf} IC$ and since, by theorem 2.4.1, $T_i \models_{\inf} CET$, it immediately follows that:
Corollary 3.3.2 If there is a safe derivation of $G_n$ from $G_0$, then

$$T \models_{int} \forall [G_0 \leftrightarrow G_n]$$

3.3.2 Affirmation soundness

Theorem 3.3.1 (Weak affirmation soundness)
If $A$ is a computed answer to $G_0$, then $A$ is a weak answer to $G_0$.

Proof: It has to be shown that conditions (A1), (A2) and (A3w) of the weak answer definition hold. Let $D$ be the disjunct of the goal $G_n$ from which $A$ has been extracted.

$A$ is a conjunction of atoms and implications by definition 3.1.7. Since the operations of the proof procedure have been exhaustively applied to the disjunct $D$ and since conclusions of implications have been replaced by false in $A$, all atoms in $A$, whether they occur as atomic subgoals or in the condition of an implication, must be suspended. So condition (A1) is satisfied.

Since $T \models_{int} \forall [G_0 \leftrightarrow G_n]$ according to corollary 3.3.2 and $G_n = D \lor Rest$, it follows that $D$ satisfies (A2). Lemma 3.1.1 has shown that omitting irrelevant implications (in the sense of definition 3.1.6) does not affect condition (A2). Recall from definition 3.1.7 that (relevant) implications in $D$ are transformed into denials by replacing their conclusions by $false$, but at the same time adding disequalities to their conditions. Adding the disequalities is justified by the fact that propagation has been exhaustively applied (so the case where the equalities hold has already been dealt with). Replacing the conclusions by false can only mean making the implications more restrictive:

$$\forall[(Cond \rightarrow false) \rightarrow (Cond \rightarrow Conc)]$$

holds for any condition $Cond$ and conclusion $Conc$. This ensures that condition (A2'), which is equivalent to (A2), will hold for $A$ if it holds for $D$.

7So, since the two conditions are equivalent, (A2) also holds, but possibly only for a different choice of Rest (i.e. not the one in $G_n = D \lor Rest$). One might thus argue that condition (A2') captures the behaviour of the proof procedure better than condition (A2).
Finally, assume ∃A is inconsistent with TC ∪ CET. Then there exists a resolution refutation of A^sk (A skolemized) from TC ∪ CET. The steps in this refutation can be mapped one-to-one to a sequence of propagation and equality rewriting steps in the proof procedure (since free variables in A are treated like skolem constants by the CPP). Hence false would have been derived by the proof procedure and logical equivalence transformations would have rewritten D to false, but then no computed answer would have been obtained from D. □

In general a computed answer may not be a good answer as the following example shows.

**Example 3.3.1**

\[ \mathcal{T}_u : p \leftrightarrow \neg q \]

\[ q \leftrightarrow r(x) \]

\[ r(a) \leftrightarrow \text{true} \]

\[ G_0 : p \]

Unfolding and logical equivalence transformation yield

\[ r(x) \rightarrow \text{false} \]

which is also the computed answer. Note that x is universally quantified over the implication, so false is implied, but cannot be derived because the condition atom r(x) is suspended. The computed answer is inconsistent with the theory \( \mathcal{T}_u \) and thus not a good answer. It is, however, a weak answer (there are no integrity constraints which could invalidate condition (A3w), and condition (A2) is trivially satisfied).

The unsoundness with respect to good answers illustrated by example 3.3.1 could be rectified in two ways — either by avoiding suspension or by providing additional integrity constraints.

The main purpose of suspension is to avoid non-deterministic search, but in the example no search takes place even if the definition of r is rewritten in homogenized form (i.e. r(x) \( \leftrightarrow x=a \)). Using the homogenized definition,
\( r(x) \) is reducible and thus false can be derived in the example so that the soundness problem is avoided.

Alternatively, if the intention was only to partially define \( r \) (so that the homogenized definition would seem undesirable), then \( r \) could have been made an external predicate without any accessible definitions, and the information that \( r(a) \) is true could have been provided by the integrity constraint true \( \rightarrow r(a) \).

Stronger soundness results can also be obtained for special cases. The following affirmation soundness theorem with respect to good answers holds for the ALP and SQO cases (cf. section 2.2.3), i.e. under the assumption that equality is the only built-in predicate ever used and that atoms of user-defined predicates are always reducible. Recall that the latter requirement can be satisfied by writing the definitions of user-defined predicates in homogenized form.

**Theorem 3.3.2 (Affirmation soundness for ALP and SQO)**

In the ALP and SQO cases, every computed answer to an initial goal \( G_0 \) is a good answer to \( G_0 \).

**Proof:** After theorem 3.3.1 it is sufficient to show that \( A \) satisfies condition \((A3_g)\).

In the ALP and SQO cases, \( A \) can contain only atoms of external predicates and of equality. Therefore, \( A \) is consistent with \( T' \cup IC \cup CET \) if and only if \( A \) is consistent with \( IC^* \cup CET \) where \( IC^* \) is obtained by exhaustively applying unfolding, equality rewriting and logical equivalence transformations to the integrity constraints. Since the integrity constraints were added to the initial goal in the first step of the proof procedure, this has also been done during the derivation, i.e. \( IC^* \) must be part of \( D \) (and thus it must be a finite set).

By arguments analogous to those used in theorem 3.3.1 to show that computed answers satisfy condition \((A3_w)\), \( A \) has to be consistent with \( IC^* \cup CET \).
and thus also with $T' \cup IC \cup CET$, i.e. $A$ satisfies condition (A3). 

If the integrity constraints provide a complete axiomatization of the theory (in the sense defined below), then the notions of weak, good, and ideal answers coincide in many cases.

**Definition 3.3.1** $IC$ is *satisfaction complete* if, for every conjunction $A$ of suspended atoms, either

$IC \cup CET \models \exists A$ or

$IC \cup CET \models \neg \exists A$.

The notion of satisfaction completeness is taken from CLP (see [JaMa94]; see also section 4.3.1.1). Just as in CLP, satisfiability is tested only for a subset of first-order formulas.

**Theorem 3.3.3** Let $A$ be a conjunction of suspended atoms. If $IC$ is satisfaction complete, then $A$ is a weak answer to an initial goal $G_0$ if and only if $A$ is a good answer to $G_0$ if and only if $A$ is an ideal answer to $G_0$.

**Proof:** Since every ideal answer is a good answer and every good answer is a weak answer in any case, it is sufficient to show that a weak answer $A$ to $G_0$ is also an ideal answer to $G_0$. So assume $A$ is a weak answer.

If $IC \cup CET \models \neg \exists A$, then $IC \cup CET \cup \exists A$ would be inconsistent, contradicting the assumption that $A$ is a weak answer.

Since $IC$ is satisfaction complete, it follows from definition 3.3.1 that $IC \cup CET \models \exists A$.

So $\exists A$ is true in all models of $IC \cup CET$ which itself is true in all intended models of $T$. Thus $\exists A$ is true in all intended models of $T$, and condition (A3) is satisfied.

Unfortunately, it is not possible to conclude that every computed answer is an ideal answer for satisfaction complete integrity constraints. This is because computed answers may contain implications, so definition 3.3.1 may not apply.
In the LP case (no suspension of user-defined predicates, no built-ins except equality, no external predicates; cf. section 2.2.3), the empty set of integrity constraints is satisfaction complete in the sense of definition 3.3.1. In the ALP, SQO and CLP cases and in the general case, the requirement that integrity constraints be satisfaction complete is not satisfied in general, but depends on the user's choice of integrity constraints, and may sometimes be impossible to satisfy. To see this, note that it would have been sufficient to show in the proof of theorem 3.3.3 that $\exists A$ is true in one intended model of $T$. The fact that $\exists A$ is true in all intended models of $T$ implies that $\mathcal{I}C$ cannot be satisfaction complete if $T$ has several intended models which disagree on the truth of (instances of) suspended atoms.

In the ALP case, the computational task may also be interpreted as identifying (or abducing) candidates for the unknown theory $T_e$ of external predicates (or abducibles). Such a theory can be obtained from a computed answer in the following way.

**Definition 3.3.2** Let $A$ be a computed answer. Remove from $A$ all implications whose condition contains an atom of a predicate other than equality and call the result $A'$. Let $E$ be the conjunction of all equalities and all implications in $A'$. Let $\sigma$ be a ground substitution of all variables in $A'$ such that

$$CET \models E\sigma$$

Let $T_e(A)$ be the theory of if-and-only-if definitions

$$p(X) \leftrightarrow X = t_1 \lor \ldots \lor X = t_n$$

where $p(t_1), \ldots, p(t_n)$ are all the atoms of the external predicate $p$ occurring as conjuncts of $A'\sigma$. If $n = 0$, then the disjunction is replaced by false. In the propositional case ($X$ empty), the disjunction is replaced by true if $p$ occurs in $A$ and by false otherwise.

The definition of $T_e(A)$ is similar to the definition of computed answer used in Fung's Iff Proof Procedure [Fu96], cf. section 4.2.2.4. An important
difference is that Fung's procedure uses an additional operation (case analysis for equality; see section 4.2.2.7) which has the effect of separating the set of implications in the goal into a set of implications containing no equality atoms in the condition and a set of denials whose conditions consist of precisely one equality atom (i.e. the denials are disequalities). See the cited sections in chapter 4 for more details.

Some notes on definition 3.3.2: $A'$ cannot contain any universally quantified variables because of equality rewrite rule (EQ6). Therefore $\sigma$ substitutes only existentially quantified and free variables. $E\sigma$ is ground and contains only atoms of the equality predicate; therefore it must be either true or false in all models of CET. For example, if $A$ is

$$p(X,Y) \land Z=c \land [ q(X) \rightarrow false ] \land [ X=a \land Y=b \rightarrow false ]$$

then $A'$ is

$$p(X,Y) \land Z=c \land [ X=a \land Y=b \rightarrow false ]$$

and $E$ is

$$Z=c \land [ X=a \land Y=b \rightarrow false ]$$

A possible choice for $\sigma$ is $(Z/c, X/a, Y/c)$, and then $E\sigma$ is

$$c=c \land [ a=a \land c=b \rightarrow false ]$$

which is true in all models of CET.

A theory $T_e(A)$ thus obtained may be regarded as an approximation to (or candidate for) the actual theory $T_e$. However, it is clear that, in the case of SQO, $T_e(A)$ cannot be expected to correspond to the EDB.

Theorem 3.3.4
If all intended models are minimal models and if the integrity constraints contain only atoms of external predicates, then

$$T' \cup T_e(A) \models_{int} IC$$

Proof: Let $M$ be an intended model of $T' \cup T_e(A)$ and let

$$C_1 \land \ldots \land C_n \rightarrow Conc$$

be an integrity constraint. Consider a ground substitution $\tau$ such that every
atom $C_i\tau$ is true in $M$. Since, by assumption, the atoms $C_i$ are external atoms and $M$ is a minimal model, $C_i\tau$ must be defined to be true in $T_\varepsilon(A)$, which implies, by construction of $T_\varepsilon(A)$, that $A$ contains a conjunct $C'_i$ such that $C_i\tau = C'_i\sigma$ for the ground substitution $\sigma$ which was used when obtaining $T_\varepsilon(A)$. Now consider two cases:

1. The atoms $C'_i$ are instances of the condition atoms of the integrity constraint. Since the $C'_i$ are atomic subgoals in the goal from which $A$ was extracted, propagation, equality rewriting and logic equivalence transformation must have been able to derive
\[
\text{true} \rightarrow \text{Conc}'
\]
with $\text{Conc}'\sigma = \text{Conc}_r$ and thus $\text{Conc}_r$ must be defined to be true in $T_\varepsilon(A)$ and thus true in $M$ (qed).

2. Otherwise, propagation would have replaced all the condition atoms in $\text{Cond}$ by equality atoms, but equality rewriting would not have been able to rewrite the condition to true. The resulting implication would have been replaced by a denial with only equalities in the condition when extracting the computed answer $A$. But since $C_i\tau = C'_i\sigma$, all these equalities must be true in $CET$ and thus yield a contradiction. But this is impossible according to definition 3.3.2. □

Note that the theorem does not hold if $T$ has non-minimal models which are intended. To see this, suppose that $p$ and $q$ are two external atoms which are both false in the minimal intended model(s) of $T$ and defined as false in $T_\varepsilon(A)$. If there is a non-minimal intended model of $T$ in which $p$ holds but $q$ does not, then the integrity constraint
\[
p \rightarrow q
\]
is not satisfied according to the propertyhood view.

The second condition of the theorem, that the integrity constraints contain only atoms of external predicates, can be relaxed in the usual way (cf. the section on SQO in chapter 4): it is sufficient to require that all atoms of
non-external predicates occurring in integrity constraints can be eliminated by unfolding. If no such restriction were imposed, then suspension might prevent the proof procedure from deriving the conclusion of an integrity constraint even if it were logically implied in \( T' \) (and thus the theorem would not hold).

### 3.3.3 Refutation soundness

It directly follows from corollary 3.3.1 that the proof procedure is refutation sound in the sense that it never rejects any good answers (and thus neither any ideal answers).

**Theorem 3.3.5 (Refutation soundness)**

If there is a safe derivation of `false` from \( G_0 \), then \( G_0 \) has no good answers.

**Proof:** If there is a safe derivation of `false` from \( G_0 \), then, by corollary 3.3.1, \[ T' \cup IC \cup CET \models \lnot[G_0 \leftrightarrow false] \]
which means that \( \exists G_0 \) is inconsistent with \( T' \cup IC \cup CET \). Hence \( G_0 \) cannot have any good answers. \( \Box \)

Theorem 3.3.5 would cease to hold if definitions in \( T \) were written in if form rather than in if-and-only-if form as the following example illustrates.

**Example 3.3.2**

Suppose that definitions in \( T \) are written in if form.

- \( T_u : p \leftarrow r \)
- \( IC : q \rightarrow p \)
- \( r \rightarrow false \)
- \( G_0 : q \)

First \( p \) is propagated and then unfolded to \( r \). Note that \( T_u \nvdash p \leftrightarrow r \).

Then `false` is generated by propagation with \( r \). However, \( q \) is consistent with \( T_u \cup IC \) and should thus be a good answer to itself. If if-and-only-if is used in \( T_u \) then \( q \) is no longer a good answer (but still a weak answer).
One might argue that theorem 3.3.5 exhibits a weakness of the proof procedure rather than a strength: the CPP never rejects a whole class of non-ideal answers (i.e. those good answers which are not ideal answers). Indeed, the proof procedure could be strengthened to reject some unwanted good answers, e.g. by evaluating atoms of user-defined and built-in predicates (such as 1<0) to false if they have no definitions. But as the discussion in examples 2.2.4 and 2.2.5 tried to show, this may also have disadvantages.

3.4 Completeness

3.4.1 Incompleteness of the general CPP

The CPP has been defined with generality in mind, not completeness. To obtain general completeness results, additional operations, which are discussed in sections 4.2.2.2 ff., have to be added to the proof procedure. At least in the ALP case, the resulting proof procedure with the added operations would look very similar to the Iff Proof Procedure of [Fu96] or SLDNFA [DeDS97] (see the comparison in the next chapter). For these proof procedures relatively strong completeness results can be proven with respect to the three-valued completion semantics of [Ku87, DeDS93] (cf. sections 2.5 and 4.2.2.2). To obtain similar results in the CALOG framework, the intended models would also have to be the models of the three-valued completion semantics.

Note that it is sensible to demand completeness only with respect to ideal answers, because if A is a non-ideal good or weak answer to G₀, then condition (A3; ) is violated which implies that

\[ T \models_{\text{int}} \forall [A \leftrightarrow \text{false}] \]

and hence condition (A2) is equivalent to

\[ T \models_{\text{int}} \forall [G₀ \leftrightarrow \text{false}] \]

i.e. the condition becomes trivial. Consequently, a non-ideal answer need not be at all related to the initial goal and thus it cannot be expected that the CPP computes such answers.
One reason why the CPP is not generally able to compute all ideal answers has been mentioned after defining computed answers (definition 3.1.7): some ideal answers whose correctness cannot be verified may be lost when extracting computed answers from a derivation. Another reason for the incompleteness of the CPP is that propagation may prevent the proof procedure from terminating, as illustrated by the following example.

**Example 3.4.1** Reconsider example 3.1.1.

\[ T_u : \text{empty} \]

\[ IC : p(x) \land q(y) \land x > y \rightarrow \text{false} \]

\[ x > y \land y > z \rightarrow x > z \]

Example 3.1.1 showed how the initial goal

\[ Go : p(a) \land q(0) \land a > 1 \]

is reduced to \text{false} in a few steps of propagation and logical equivalence transformation.

Now consider instead the initial goal

\[ G'_0 : p(a) \land q(2) \land a > 1 \]

which is consistent with the definition of “\( > \)” and the integrity constraints. In fact, \( G'_0 \) is an ideal answer to itself.

Recall from example 3.1.1 (substituting 2 for 0) that propagation first generates the two “useful” resolvents

\[ a > 2 \rightarrow \text{false} \]

\[ 1 > z \rightarrow a > z \]

which, when resolved with each other, now yield

\[ 1 > 2 \rightarrow \text{false} \]

This time \text{false} cannot be derived and the proof procedure should terminate successfully (in example 3.1.1 the resolvent was \( 1 > 0 \rightarrow \text{false} \), yielding a contradiction after unfolding \( 1 > 0 \)).

However, there are still numerous ways of applying propagation between the derived implications and the original integrity constraints. For example,
propagation between the implication \( A > 2 \rightarrow \text{false} \) and the transitivity integrity constraint generates
\[
A > Y \land Y > 2 \rightarrow \text{false}
\]
which can again be used with transitivity to yield
\[
A > Y' \land Y' > Y \land Y > 2 \rightarrow \text{false}
\]
and then in turn
\[
A > Y'' \land Y'' > Y' \land Y' > Y \land Y > 2 \rightarrow \text{false}
\]
and so on, ad infinitum.

Several ways to address the non-termination problem by restricting propagation are discussed in section 5.2.

### 3.4.2 The negation-free LP case

Since a general completeness result cannot be obtained as example 3.4.1 has shown, it seems useful to at least establish a completeness result for a special case. Recall that in the LP case there are no external predicates, equality is the only built-in predicate, and there is no suspension of user-defined predicates.

Assume further that there is no negation in the bodies of definitions or in the initial goal. Since there are neither any integrity constraints in the LP case, implications can never appear in the goal and thus the proof procedure will never use the propagation operation. The following theorem shows that, under these strong restrictions, the proof procedure can be understood as an implementation of SLD resolution.

**Theorem 3.4.1 (Correspondence between SLD and CPP)**

In the negation-free LP case, let \( G_0 \) be an initial goal and let \( \mathcal{P} \) be the Horn clause program which corresponds to \( T \) and is obtained by rewriting definitions in \( T \) into clauses in the way defined in definition 2.1.2. Assume that
\[
(\leftarrow R_0, \sigma_0), \ldots, (\leftarrow R_{n-1}, \sigma_{n-1}), (\leftarrow, \sigma_n)
\]
is an SLD refutation of \( R_0 = G_0 \) from \( \mathcal{P} \) with answer substitution \( \sigma_n \) (\( \sigma_0 \) being the empty substitution). Let \( A \) be the conjunction of equalities defined by \( \sigma_n \).
Then there exists a CPP derivation such that $A$ is a computed answer to $G_0$ obtained from the derivation (up to commutativity).

**Proof:** By induction on $n$. The case $n = 0$ is trivial ($G_0 = \text{true}$, $\sigma_0$ is empty, $A = \text{true}$).

Assume that $G_0, \ldots, G_i$ is a CPP derivation such that $G_i = (R_j \land Eq_j) \lor Rest$ (for some $0 \leq j < n$ and some formula $Rest$) where $Eq_j$ is the conjunction of equalities defined by the substitution $\sigma_j$ in the SLD refutation.

In the SLD refutation, an atom $C$ in the conjunction $R_j$ is resolved with a clause

$$H \leftarrow D$$

in $\mathcal{P}$ which has been obtained by rewriting a definition

$$H \leftarrow D_1 \lor \ldots \lor D_n$$

in $\mathcal{T}$, i.e. $D = D_l$ for some $l \in \{1, \ldots, n\}$.

Since definitions in $\mathcal{T}$ are in homogenized form in the LP case, there must be a substitution $\sigma$ such that $H\sigma = C$. $R_{j+1}$ is obtained by replacing $C$ by $D'\sigma'$, where $D'$ is obtained by omitting all equality atoms from $D$ and $\sigma'$ is obtained by integrating these equality atoms into the substitution $\sigma$. Then $\sigma_{j+1} = \sigma_j\sigma'$.

Since $H\sigma = C$, the atom $C$ in the CPP goal $G_i$ is reducible and can be unfolded to $(D_1 \lor \ldots \lor D_n)\sigma$. The disjunction can be split (several times, if necessary), and equality rewriting can be applied to the disjunct containing $D_l\sigma$. Since equality rewriting is a sound and complete implementation of the unification algorithm, the resulting disjunct of the goal in the CPP derivation must be equal to $(R_{j+1} \land Eq_{j+1})$ and the goal is equal to

$$(R_{j+1} \land Eq_{j+1}) \lor Rest'$$

for some formula $Rest'$.

If $j + 1 = n$, then $R_{j+1}$ is the empty goal and thus the new goal is $Eq_{j+1} \lor Rest'$. No further CPP operations can be applied to $Eq_{j+1}$ and since $Eq_{j+1}$ does not contain any implications, it is a computed answer. It follows
that \( Eq_{j+1} \) is equal to \( A \) (up to commutativity).

Now assume that the minimal Herbrand model of the logic program \( \mathcal{P} \) corresponding to the theory \( \mathcal{T} \) is the only intended model of \( \mathcal{T} \) and that all ideal answers are conjunctions of equalities (i.e. substitutions). Since SLD resolution is complete with respect to the minimal Herbrand model semantics, we get:

**Corollary 3.4.1 (Completeness for LP case)**
Let \( A \) be an ideal answer to \( G_0 \) in the negation-free LP case. Then there exists a CPP derivation of \( A \) from \( G_0 \).

### 3.4.3 The general case

Given that propagation may not terminate, the best one might hope to show for the unrestricted CPP is that, for a given ideal answer, at some point in a derivation a computed answer could be derived (if the derivation did terminate) from one of the disjuncts of the current goal which corresponds to or at least approximates the ideal answer in some sense. Thus the following definition may prove useful when trying to approximate completeness results.

**Definition 3.4.1** \( A \) is a computed pre-answer to the initial goal \( G_0 \) if there exists a derivation of a goal \( G_n \) from \( G_0 \) such that \( G_n = D \lor \text{Rest} \), where \( D \neq \text{false} \) and no CPP operation except for propagation can be applied to \( D \) and \( A \) is the result of extracting an answer from \( D \), as in definition 3.1.7 of computed answer.

A computed pre-answer \( A \) extracted from a goal disjunct \( D \) may not be a weak (semantic) answer because propagation may still be able to derive \text{false} in \( D \). However:
Lemma 3.4.1 If $A$ is a computed pre-answer to $G_0$ then $A$ satisfies conditions (A1) and (A2) of the semantic answer definition.

Proof: Unfolding has been exhaustively applied to a computed pre-answer, so condition (A1) is satisfied for the same reasons as for computed answers. The fact that condition (A2) is satisfied follows from corollary 3.3.2 in the same way as in the proof of theorem 3.3.1.

The proof procedure cannot be expected to generate a given ideal answer $A$ but at best a computed pre-answer $A'$ such that

$$T \models_{int} \forall [A' \leftarrow A].$$

Since

$$T \models_{int} \forall [G_0 \leftarrow A']$$

holds according to lemma 3.4.1, $A'$ can be viewed as an approximation to the ideal answer $A$.

In the general case one cannot expect to obtain completeness results independently of the choice of intended model (which was fixed in the negation-free LP case), because the proof procedure cannot be both affirmation sound and affirmation complete with respect to two semantics which have different sets of ideal answers. In fact, the proof procedure can be complete only for the semantics with the smallest set of computed answers if it is to be sound with respect to all semantics. A good candidate for a semantics with a smallest set of computed answers is the three-valued completion semantics which is used in [Fu96] and [DeDS97] for the related proof procedures, Iff and SLDNFA (see especially the discussion in section 11.2 of [DeDS97]).

Now reconsider the proof of theorem 3.4.1. There a CPP derivation corresponding to a given SLD refutation of $\leftarrow G_0$ was constructed. In the general case, some steps in the SLD refutation may correspond to the unfolding of suspended atoms, and it may thus no longer be possible to construct a CPP derivation corresponding to a given SLD refutation. However, since SLD is
complete independently of the selection rule used in the derivation, it is possible to delay the selection of suspended atoms as long as other resolution steps are possible.

But in the general case it is not always possible to use SLD to obtain an SLD refutation corresponding to a given CPP derivation, because negative literals may appear in the goal. And SLDNF, the extension of SLD which can handle negative literals, is not complete itself since it cannot select non-ground negative literals. As was pointed out in section 3.1.3, this problem of SLDNF, which is known as the problem of constructive negation, is partly solved by the CPP, i.e. the CPP sometimes returns a computed answer where SLDNF would flounder.

Therefore it is not clear whether the completeness theorem obtained for the LP case can be generalized. It may be possible to establish a more formal correspondence between computed pre-answers and ideal answers, but the theoretical and practical value of such a result appears to be of little value, especially considering that stronger results are already known for the related proof procedures. For these reasons, a formalization is not attempted here, but left open for possible future research.
Chapter 4
Unified Areas

This chapter compares the CALOG framework, its semantics and its proof procedure to related work in the three main areas which the framework seeks to unify — SQO, ALP and CLP. It is shown how and to what extent semantics and procedures proposed in these areas correspond to the CALOG framework's semantics and proof procedure as defined in chapters 2 and 3. The comparison aims to further contribute to the unification of the different areas by presenting them all in a similar way, thus emphasizing common features, especially in the semantics, which are usually hidden under framework specific terminology.

The chapter also contains a brief discussion of concurrent LP and concurrent CLP whose incorporation into the unified framework may provide interesting opportunities for future research.

4.1 Semantic Query Optimization

Optimizing a query posed to a database means rewriting the query into an equivalent one that can be executed more efficiently, i.e. whose answers are the same as those of the original query, but can be retrieved more quickly. The area of database query optimization can be divided into syntactic strategies, which exploit certain properties of the database and of the (relational) operators occurring in the query, and semantic strategies, which use semantic knowledge about the database to transform the query. Semantic knowledge is usually
provided in the form of integrity constraints, and the semantic strategies which make use of them are referred to as SQO.

### 4.1.1 Deductive databases and SQO semantics

A logic-based approach to SQO in deductive databases is described by Chakravarthy, Grant and Minker in [ChGrMi90, ChGrMi88]. A deductive database consists of an extensional and an intensional part. In [ChGrMi90], the intensional database (IDB), which corresponds to the user-defined part \( T_u \) of the theory in the CALOG framework, is restricted to non-recursive Horn clauses without function symbols, and integrity constraints must not be recursive either. These restrictions have been partly lifted in further work on SQO (see [GaLo93] and the references given there).\(^1\)

The extensional database (EDB), which corresponds to the inaccessible theory \( T_e \) in the CALOG framework, is assumed to be an enumeration of ground facts. The aim of SQO is to transform a given query, without accessing the EDB, into an equivalent query which consists only of EDB (and possibly built-in) atoms and satisfies the integrity constraints. As has already been mentioned in chapter 1, this is very similar to the abductive task in ALP (if the EDB predicates are identified with the abducibles) and is a special case of the computational task in the CALOG framework of reducing the initial goal to an equivalent goal consisting only of suspended atoms (assuming for the SQO case that user-defined definitions are written in homogenized form so that they are never suspended). While the literature on SQO concentrates on defining procedures for the actual query optimization, it may be useful to also formalize the semantics, especially for a comparison with the CALOG framework.

Given an initial goal (or query) \( Q_0 \), the query \( Q \) is an optimized query (or “answer”) if

\(^1\)The paper [GaLo93] also extends the approach of [ChGrMi90] to deal with negative literals and IDB clauses with disjunctive heads.
(1) $Q$ consists of atoms of EDB predicates only

(2) $\text{IDB} \cup IC \models \forall [Q_0 \leftrightarrow Q]$

(3) $\text{IDB} \cup IC \cup \exists Q$ is consistent

Condition (1) corresponds to condition (A1) in the CALOG framework's semantics. Condition (2) corresponds to condition (A2') with $Q$ taking the place of $A \lor \text{Rest.}$

Condition (3) corresponds to the good answer condition (A3;). It can be considered an optional part of the SQO semantics as the optimized query will be passed to the EDB in any case (and thus it is not mandatory that inconsistencies with integrity constraints be detected during the query optimization).

The analogue of the ideal answer condition (A3;i), which might be expressed as

$$\mathcal{DB} \models_{\text{int}} \exists Q$$

where $\mathcal{DB}$ is the whole database (IDB and EDB together), cannot be expected to hold, as the optimized query $Q$ may not have any answers in the EDB (which is acceptable in SQO). Moreover, while in the CALOG framework the construction of candidate theories $\mathcal{T}_e^*$ (cf. theorem 3.3.4) to approximate the actual theory $\mathcal{T}_e$ of external predicates is of some interest (especially in the ALP case), it is clearly not the aim of SQO to reconstruct the EDB.

4.1.2 Procedures for SQO

The two-phased compiled approach of [ChGrMi88, ChGrMi90] consists of a compilation phase and a transformation phase. The different steps in these phases are sketched in the following and compared, as far as possible, to corresponding operations in the CALOG proof procedure. As the procedures proposed for SQO are geared towards the needs of (special types of) deductive databases, they may prove efficient for them and serve as suggestions to

---

\footnote{So whereas a goal in the CALOG framework can have infinitely many answers, here the optimized query is a (finite) disjunction. This is possible because there is no recursion.}
improve the efficiency of the CALOG proof procedure for specific database applications. On the other hand, the CPP is more general, applies to wider classes of databases (and other applications, of course) and may help to generalize and simplify the SQO procedures.

4.1.2.1 The compilation phase

[ChGrMi88] specifies an algorithm which first transforms a given database into a structured one in which all predicates are either purely intensional or purely extensional and where integrity constraints contain only EDB atoms. While the disjointness of intensional and extensional predicates (or user-defined and external predicates in the CALOG framework) is easy to achieve by introducing auxiliary extensional predicates, the algorithm to eliminate intensional predicates from integrity constraints relies on the assumption mentioned above that IDB clauses are non-recursive, and the algorithm may not terminate without this assumption.

The aim of the compilation phase is to generate “semantically constrained axioms” which combine the information given in the IDB and the integrity constraints. For this purpose the IDB clauses are compiled into axioms by replacing any IDB atoms in their bodies by their definitions (i.e. by unfolding them). If one atom is defined by several IDB clauses, then several axioms are obtained as a result. In addition, one axiom \( e(X) \leftarrow e(X) \) is added for every EDB predicate \( e \).

Example 4.1.1 Let \( p \) and \( q \) be IDB predicates and let \( a \) and \( b \) be EDB predicates of a structured database. Let the IDB consist of the clauses

- \( p(X) \leftarrow q(X) \)
- \( q(X) \leftarrow a(X) \)
- \( q(X) \leftarrow b(X) \)

The following set of axioms is obtained:

- \( p(X) \leftarrow a(X) \)
Next, the integrity constraints are expanded: the parameters of all their atoms are replaced by fresh variables and additional equalities are introduced as conditions to represent the actual parameters. For example,

\[ a(1) \rightarrow \text{false} \]

is expanded to

\[ a(X) \land X=1 \rightarrow \text{false} \]

Note that applying the equality rewrite operation (definition 3.1.4) of the CA-LOG proof procedure to an expanded integrity constraint restores the original, unexpanded integrity constraint (modulo variable renaming). Integrity constraints are expanded to prepare them for the next step in the compilation, which is called partial subsumption: an integrity constraint \( I \) partially subsumes an axiom \( A \) if a proper, non-empty subset of (the literals in) \( I \) subsumes \( A \). For the purposes of this discussion, a test for partial subsumption can be most easily understood as checking whether resolution (and thus propagation) between the expanded integrity constraint and the axiom is possible. For example, the expanded integrity constraint

\[ a(X) \land X=1 \rightarrow \text{false} \]

partially subsumes each of the three axioms from example 4.1.1 with body \( a(X) \). If an integrity constraint partially subsumes an axiom, then the residue of the integrity constraint, which is that part of the integrity constraint which does not match any atoms in the body of the axiom, is attached to the axiom to yield a semantically constrained axiom. In the example, the first axiom,

\[ p(X) \leftarrow a(X) \]

becomes
p(X) ← a(X) \{ X=1 → false \}
where the part in braces is the attached residue. The declarative reading of this semantically constrained axioms is
p(X) ← a(X) \land X \neq 1
The other two partially subsumed axioms from example 4.1.1 are transformed analogously. In case several integrity constraints partially subsume one axiom, their residues are also attached to the same axiom to further semantically constrain it.

After finishing the compilation phase (which also includes some final steps of simplifying the semantically constrained axioms which shall be omitted here), the information represented by the semantically constrained axioms is equivalent to the information of the original IDB together with the integrity constraints. In the terminology of the CALOG proof procedure, after some steps of unfolding (to generate the axioms), “reverse equality rewriting” (to expand the integrity constraints) and partial subsumption together simulate several steps of propagation which will not have to be applied again, possibly repeatedly, in the second phase, the query transformation. Whether this yields a computational advantage or disadvantage (due to the preprocessing overhead which also compiles clauses and integrity constraints which might otherwise never be invoked) depends on the application (i.e. the database and the query to be optimized). It has to be stressed again that for this compilation to be feasible the IDB clauses have to be free of recursion, so a general adaptation of these techniques in the CALOG framework is not possible. Rather, the more general operations of the CALOG proof procedure could be used to perform SQO on recursive databases.

4.1.2.2 The transformation phase

After the compilation phase, the first step in the transformation phase is called query and residue modification. It means using the semantically constrained axioms as definitions used to unfold query atoms. The necessary substitu-
tions (to make the head of the axiom and the query atom unify) are applied to the body of the semantically constrained axiom and any residues (of integrity constraints from the first phase) attached to it (thus residues may also be modified). The body of the axiom, with the residues still attached to it, becomes part of the new semantically constrained query. The residues may be rearranged so that they all appear at the end of the resulting query, using the same syntax as for semantically constrained axioms. If more than one semantically constrained axiom unifies with a given query atom, then a disjunction of two semantically constrained subqueries is introduced (which are handled separately from then on, i.e. as if the disjunction is split immediately).

In the CALOG framework, the query modification corresponds to unfolding (plus splitting, if needed), and the residue modification is achieved by equality rewriting. As [ChGrMi90] notice, query and residue modification may lead to built-in expressions (equalities, inequalities, arithmetic operators etc.) becoming fully instantiated and thus evaluable. The CALOG framework’s definitions of built-in predicates and suspension (of not only built-ins but of all predicates) constitute a formalization and generalization of these concepts.

Evaluation of built-in atoms may lead to residues being reduced to the empty clause (or, depending on the chosen method of representation, to the implication true → false). This means that the query or subquery has no answers and can be reduced to false. E.g., continuing example 4.1.1, if the initial query is

\[
p(1)
\]

then query and residue modification (using the semantically constrained axiom of the previous subsection) yield the disjunction

\[
( a(1) \{ 1=1 \rightarrow \text{false} \} ) \lor b(1)
\]

The first disjunct (or subquery) contains an evaluable atom which can be reduced to true, yielding the implication true → false in the residue. Thus the query can be reduced to the second disjunct, b(1).

Modified queries may become complicated due to the residues attached
to them, and [ChGrMi90] propose to simplify them by removing all those residues which are irrelevant to the modified query. In the worst case, this may lead to more work being invested in undoing superfluous work that was done when compiling the integrity constraints in the first phase. In the CALOG framework, non-applicable integrity constraints would simply not be used for propagation. Moreover, for the restricted types of databases considered by SQO, the hyper-resolution strategy discussed in chapter 5 could be used to ensure that an integrity constraint is applied only if all of its conditions can be shown to hold, thus avoiding the generation of residues which cannot be further used.

The second and last (major) step in the query transformation phase uses the residues to derive further information which may prove useful in answering the query (when passing it to the EDB). Note that any such computations are optional in the same sense as condition (3) of the given semantics for SQO is optional — it is not strictly necessary to detect inconsistencies between the query and the residues.

Chakravarthy, Grant and Minker distinguish four different types of residues and treat them separately.

The first is the empty clause (or $\text{true} \rightarrow \text{false}$), i.e. the (sub)query leads to a contradiction and has no answers.

The second is a denial residue of the form $\text{Cond} \rightarrow \text{false}$. In this case, if the residue subsumes the query, then the query does not have any answers (in CALOG terminology, $\text{Cond}$ can be resolved away by propagation, yielding again $\text{true} \rightarrow \text{false}$).

The third is a "unit clause", i.e. an implication of the form $\text{true} \rightarrow \text{Conc}$. The conclusion, $\text{Conc}$, could be added to the query as auxiliary information. However, if it (or an instance of it) already occurs as an atom in the query, then it could also be deleted from the query since the residue implies that the query atom will simply succeed.

The fourth and last type is a general implication $\text{Cond} \rightarrow \text{Conc}$. If the
atoms in \( \text{Cond} \) occur in the query, then the conclusion, \( \text{Conc} \), can be added as auxiliary information which may prove helpful in answering the query.

The CALOG proof procedure's more general approach provides a uniform treatment of all four types of residues by propagation. The proposed alternative usage of the unit clauses for deletion of query atoms corresponds to the deletion operation proposed as an extension in chapter 5. Again, the more general approach of the CALOG framework shows that deletion need not be restricted to the special case of unit clauses, but can also be applied in the fourth case of a general implication.

When extracting the optimized query at the end of the second phase of optimization, the residues attached to the final semantically constrained query can be discarded since the optimized query will be answered by accessing the EDB. Condition (2) of the semantics guarantees that the union of the answers to all disjuncts of the optimized query is the set of all answers to the initial query.

4.2 Abductive Logic Programming

4.2.1 Declarative semantics

As already outlined in chapter 1, an ALP framework is defined (cf. [KaKoTo93, To95]) as a triple \((T, Ab, IC)\) where \(T\) is a normal logic program, \(Ab\) a set of predicate symbols (the abducibles) and \(IC\) a set of integrity constraints. Given an observation or goal \(G_0\), the abductive task is to find an abductive hypothesis or abductive explanation (or simply answer) \(\Delta\) for \(G_0\) which satisfies the following three conditions:

(1) \(\Delta\) is a conjunction of ground abducible atoms

(2) \(T \cup \Delta \models G_0\)

(3) \(T \cup \Delta\) satisfies the integrity constraints \(IC\)
Condition (1) is a special case of condition (A1) in the CALOG framework’s semantics, provided that only external predicates are suspended (which can again be achieved by writing user-defined definitions in homogenized form; moreover, ALP frameworks usually do not have built-in predicates apart from equality). In some ALP frameworks not only ground atoms, but also negative literals or even full first-order sentences are permitted in $\Delta$ (see the comparison with SLDNFA below).

Condition (2) could also be written as
\[ T \models (\Delta \rightarrow G_0) \]
which is very similar to condition (A2').

Condition (3) may be interpreted as corresponding to one of the (A3) variants, depending on which definition of integrity constraint satisfaction and, in the case of (A3'), which notion of intended model is used.

A general discussion of the different notions of integrity constraint satisfaction has already been conducted in section 2.4. There it has also been pointed out that the approach taken by [KaMa90] resembles the propertyhood view taken in the CALOG framework. [KaMa90] define generalized stable models as follows:

Let $(T, Ab, IC)$ be an abductive framework and $\Delta$ a set of ground abducible atoms. Then $M(\Delta)$ is a generalized stable model for the abductive framework if

- $M(\Delta)$ is a stable model of $T \cup \Delta$
- $M(\Delta) \models IC$

For comparison, recall from the remarks after definition 2.4.1 that the propertyhood view could be written in the form

\[ T' \cup \Delta \models_{\text{int}} IC \]

where $\Delta$ represents the external theory $T_e$. This is very similar to [KaMa90] if the intended models are precisely those stable models in which the integrity
constraints are true. It follows that if $M(\Delta)$ is a generalized stable model for an abductive framework and furthermore

$$M(\Delta) \models G_0$$

then $\Delta$ is an abductive explanation for $G_0$ in the sense of conditions (1)–(3) above.

In addition to conditions (1)–(3), abductive explanations are usually required to be basic, i.e. not further reducible. For example, given the theory

$$p \leftarrow q$$
$$q \leftarrow r$$

$q$ is an explanation for $p$ which is not basic whereas $r$ is a basic explanation. The condition is trivially satisfied if abducibles have no definitions (because then, in the example, $q$ could not be an abducible and would violate condition (1) of the semantics). The notion of suspension in the CALOG framework may be seen as a generalization of the concept of basic explanations — if atoms of predicates other than external predicates become part of an answer because they are suspended, then the explanation is still basic in the sense that it is not further reducible (according to the framework definition of reducible as unsuspended).

It is thus not necessary in the CALOG framework to require that abducibles (i.e. external predicates) have no definitions, allowing some interesting ALP applications in which abducibles are partially defined. For example, in the event calculus the happens predicate may be partially predefined while some of its instances may still have to be abduced.

A further requirement frequently imposed on abductive explanations is minimality. An abductive explanation $\Delta$ is called minimal if no proper sub-conjunction of $\Delta$ is an abductive explanation, i.e. the generation of hypotheses which are not needed to explain an observation is to be avoided. There are other possible definitions of minimality and [DeDS97] list some of them. For example, if assuming either $A$ or $B$ is required to obtain an abductive explanation, but $B$ also requires the additional assumption $C$ while $A$ does not, then
one could say that the explanation $A$ is "more minimal" than the explanation $B \land C$.

The issue of minimality will again be addressed at several points in the following comparison between the CPP and other abductive proof procedures (see especially example 4.2.7).

### 4.2.2 Proof procedures

Whereas the CALOG framework and its semantics can be seen as drawing about equally from ALP, CLP and SQO, its proof procedure, the CPP, is most closely related to and in fact in large parts based on two proof procedures previously defined for ALP, namely Fung's Iff Proof Procedure (IPP) [Fu96, FuKo96] and Denecker and DeSchreye's SLDNFA [DeDS92, DeDS97]. The CPP could be regarded as a generalization or abstraction of these two proof procedures, and the following sections compare the procedures in some detail to point out similarities, differences and possible extensions of the CPP based on additional operations which the other two procedures include.

Comparisons with other related abductive proof procedures have already been conducted in [DeDS97, Fu96]. Two of them shall be briefly mentioned here.

The procedure of [CoThTo91] is noteworthy for its use of if-and-only-if definitions. These definitions are used for the procedure's only operation, which corresponds to unfolding. Quantifiers are represented explicitly and no restrictions are imposed on their nesting so that computed answers, called explanation formulas, may get rather complicated and, as [DeDS97] point out, require a separate satisfiability check.

Teusink's abductive extension of SLDFA resolution [Dr93] is also interesting as he may have been the first to allow variables in abductive explanations [Te93] (thus solving the floundering abduction problem\(^3\) — note that the CALOG framework also solves this problem). The procedure he defines includes

\(^3\)Traditional abductive proof procedures cannot select a non-ground abducible atom.
an implementation of constructive negation (cf. section 3.1.3), thus avoiding floundering altogether. Recently, [Te96] proposed to use Kunen’s three-valued logic to define a three-valued completion semantics for his abductive extension of SLDFA, yielding an abductive framework rather close to Fung’s and De-Necker and DeSchreye’s (and hence also similar to one instance of the CALOG framework).

4.2.2.1 Underlying frameworks

In the underlying framework of the IPP, knowledge is represented, just as in the CALOG framework, both by definitions and by integrity constraints. The definitions were originally (in [Fu96]) written in if form, but since disjunctions were allowed in the bodies of definitions and since the definitions had to be in homogenized form, the difference was mainly syntactic (and, moreover, the completion of the definitions was used for the semantics). [FuKo96] now uses if-and-only-if definitions (which still have to be homogenized).

In SLDNFA, on the other hand, only normal logic programs with if clauses are permitted (no disjunctions in the body, several clauses per predicate) and integrity constraints are not represented directly. [DeDS97] point out that integrity constraints can be transformed into a normal logic program using the transformations of [LiTo84]. For example, the integrity constraint

\[ \text{manager}(X) \rightarrow \text{bonus}(X) \]

can be represented by

\[ \neg p \]

where \( p \) is a new predicate defined by

\[ p \leftarrow \text{manager}(X) \land \neg \text{bonus}(X) \]

However, the reading and intended operational behaviour of the original integrity constraint is probably more obvious than that of the transformed pair of sentences.

The IPP requires definitions and integrity constraints to be allowed (i.e. range-restricted, cf. definition 3.1.9) in order to facilitate the proof of some
results. SLDNFA uses instead a safety requirement which leads to floundering if only non-ground negative literals with certain types of variables can be selected (see [DeDS97] for the precise definition, which is rather complicated; the CALOG framework allows the corresponding definition 3.1.8 used by the CPP to be much simpler, although it basically has the same effect).

Both the IPP and SLDNFA distinguish between two classes of predicates: defined (non-abducible) predicates, which correspond to the user-defined predicates in the CALOG framework, and abducible predicates, which correspond to the external predicates. In addition, equality plays a special role and is effectively treated as the only built-in predicate.

4.2.2.2 Semantics

While formalized in rather different ways, the semantics of the IPP and SLDNFA can both be considered variants of the general ALP semantics specified in section 4.2.1 and are thus similar both to each other and to the ALP instance of the CALOG framework’s semantics. They are both based on a three-valued completion semantics originally introduced by Kunen [Ku87] (see also [DeDS93]). As has already been mentioned in section 2.5, the three-valued completion semantics is weaker than the other semantics discussed in that section in the sense that the set of models under the three-valued completion semantics is a superset of the set of models under any of the other semantics [DeDS97, theorem 2.2]. This implies that requiring a sentence to be true in all models of the three-valued completion semantics is a stronger condition than for any of the other semantics, i.e. fewer sentences satisfy this condition and hence the task of computing all such sentences as answers to a goal (and thus to achieve completeness results) is easier.

For the IPP, let $T$ denote the (if-and-only-if) theory of defined predicates including a definition for equality. Then $A$ is an answer to the initial goal $G_0$ if
(IPP1) $A$ is a conjunction of ground abducible atoms

(IPP2) $T \cup \text{Comp}(A) \models_3 G_0$

(IPP3) $T \cup \text{Comp}(A) \models_3 IC$

Condition (IPP1) is the same as condition (1) of the general ALP semantics. However, the IPP does not flounder when only non-ground abducible atoms are left — rather, free and existentially quantified variables in goal atoms are instantiated when extracting a computed answer from a derivation.

Conditions (IPP2) and (IPP3) require both the initial goal and the integrity constraints to be theorems, in Kunen's three-valued logic, of the completion of the combined theory $T \cup A$.

(IPP3) thus expresses a 3-valued variant of the theoremhood view of integrity constraint satisfaction which is one possible interpretation of condition (3) of the general ALP semantics from section 4.2.1. It is similar (but not identical) to the CALOG framework’s good answer condition (A3$_g$) combined with the propertyhood condition $T \models_{int} IC$. The differences between the theoremhood and propertyhood views of integrity constraint satisfaction have been discussed in section 2.4.

Contrary to some other three-valued logics, in Kunen's version [Ku87], which is used here, an equivalence (and hence an if-and-only-if definition) can only take on the truth values $\text{true}$ and $\text{false}$: $A \leftrightarrow B$ is true if $A$ and $B$ have the same truth values and false otherwise. So undefined $\leftrightarrow$ undefined has the truth value $\text{true}$. This has the effect that the use of "$\models_3$" in conditions (IPP2) and (IPP3) rules out certain $A$ which would qualify as answers (but are not computed by the IPP) if two-valued logic was used instead. This is illustrated by the following example.

Example 4.2.1 (Difference between 2- and 3-valued logic)

\[
T : p \leftrightarrow q \lor \neg q \\
q \leftrightarrow q
\]
Then $T \models p$, i.e. $A = \text{true}$ would be an answer to $G_0$ in two-valued logic, but $T \not\models_3 p$ because there is a 3-valued model of $T$ in which both $p$ and $q$ are assigned the truth value undefined.

Consequently, $A = \text{true}$ is not an answer according to conditions (IPP1)–(IPP3), and the IPP may go into a loop when processing the query $p$ without being incomplete.

Owing to the use of the three-valued completion semantics and the inclusion of additional operations in the IPP not present in the CPP (see the following sections), [Fu96] is able to prove a strong affirmation completeness result for the IPP: every minimal semantic answer is computed in a finite branch of a derivation starting from $G_0$ (IPP derivations and their operations are described and discussed below). A similarly strong result could be obtained for the CPP (in the ALP case and for the three-valued completion semantics) if the additional operations were added to the CPP.

The IPP is also affirmation and refutation sound with respect to its three-valued semantics, i.e. every computed answer to $G_0$ is a semantic answer to $G_0$ and if all branches of a derivation starting from $G_0$ end in $\text{false}$, then $T \models_3 \forall \neg G_0$.

Furthermore, as remarked above, using two-valued logic weakens conditions (IPP2) and (IPP3), and consequently the soundness results (but not the completeness results) continue to hold if the two-valued completion semantics is used instead, i.e. if “$\models_3$” is replaced by “$\models$” in (IPP2) and (IPP3).

To specify the semantics for SLDNFA, let $T$ be $\text{Comp}(P, \text{NonAb}) \cup CET \cup Abd2$ where $\text{Comp}(P, \text{NonAb})$ denotes the selective completion (only completing non-abducibles) of the normal logic program $P$ defining the non-abducible predicates and where $Abd2$ is the set of formulas $\forall X \ [ a(X) \lor \neg a(X) ]$ for all abducible predicates $a$ (with $X$ being a tuple of variables). Then $A$ is an answer to the initial goal $G_0$ if
(SLDNFA1) $A$ is a formula of first-order logic containing only equality and abducible atoms

(SLDNFA2) $T \models_3 \forall[G_0 \leftarrow A]$

(SLDNFA3) $T \not\models_3 \exists A$

This semantics is even closer to the CALOG framework’s semantics than the IPP’s. [DeDS97] also define an alternative semantics in which condition (SLDNFA1) is replaced by a condition similar to (IPP1). The CALOG framework’s condition (A1) which allows non-ground atoms and denials, may be seen as a compromise between the two. See also section 4.2.2.4 which compares the ways computed answers are extracted from a derivation.

4.2.2.3 Operations corresponding to CPP operations

Although they are again formalized in rather different ways, SLDNFA and the IPP use very similar operations. The four CPP operations form a subset of these operations and it is possible to extend the CPP to also include the additional operations defined for the IPP and SLDNFA which are discussed in the following sections. One reason why this has not been done is to avoid a specialization of the CPP which is geared towards abduction. A more general proof procedure leaves open the possibility of extending and specializing it in different ways. Another reason is the fact that answers generated when using the additional operations are sometimes not minimal (basically, there is a trade-off: one either gets not all the answers or too many of them).

As for differences in formalization, the IPP’s is very close to the CPP’s, but the operations of SLDNFA are defined in a rather different, more complicated way, partly due to the fact that if-and-only-if is not used in definitions and that integrity constraints and implications are not represented explicitly. Fung’s work [Fu96] deserves credit for considerably simplifying Denecker and DeSchreye’s original approach [DeDS92] and this thesis hopes to have made
further contributions towards a generalization and simplification of both of the other two proof procedures.

The unfolding operations in the CPP (see definition 3.1.2) and IPP are very similar. However, the IPP assumes that definitions in $T$ are written in homogenized form so that exactly the abducibles are suspended. The more general form of suspension which is possible in the CALOG framework has not been considered in the IPP's underlying framework.

SLDNFA has instead an operation called positive resolution and uses normal logic program clauses, i.e. there may be several steps of positive resolution applicable at one point in a derivation (no concept of suspension exists) and the alternatives give rise to different derivations rather than to disjunctions being introduced explicitly into the goal. This precludes the possibility of extending the procedure by the CPD techniques for propagation across disjunct boundaries (see next chapter).

Propagation in the IPP is only performed between an atomic subgoal and an implication, i.e. the $\text{Sus}$ part of the second implication in definition 3.1.5 is empty. This is justified by the fact that the IPP never suspends atoms of non-abducible predicates.

Example 4.2.2 It is not possible in the IPP to apply propagation to the two implications

\[ A > 0 \rightarrow \text{false} \quad \text{and} \]
\[ 1 > Z \rightarrow A > Z \]

of example 3.1.1 to generate the implication

\[ 1 > 0 \rightarrow \text{false} \]

which has a reducible condition (and thus introduces $\text{false}$ into the goal). The IPP has no built-in predicates and thus does not "know" that $1 > 0$ evaluates to $\text{true}$. If the user defines "\( > \)" or adds an integrity constraint $\text{true} \rightarrow 1 > 0$, then the second implication will eventually propagate $A > 0$ into the goal which can then be used with the first implication to derive $\text{false}$. 
As a consequence, propagation is more restricted in the case of the IPP and hence potentially more efficient. See also section 5.2.

SLDNFA defines a negative resolution operation instead of propagation. Similarly to the IPP, SLDNFA does not resolve two implications with each other.

In contrast to the CPP and the IPP, which both represent equalities explicitly and deal with them using rewrite rules based on the unification algorithm defined in [MaMo82] (see definition 3.1.4), SLDNFA instead computes solved forms of equalities (idempotent substitutions) whenever applying a step of positive or negative resolution (i.e. unfolding or propagation). However, in the case of a negative resolution step, the part of the solved form assigning terms to positive variables (essentially the free variables from the initial goal and existentially quantified variables introduced by unfolding) is kept as an explicit equality in the goal and the rest of the substitution is stored separately. In the end, this has the same effect as equality rewriting in the CPP and IPP, but it is more complicated.

4.2.2.4 Extraction of computed answers

It is useful to examine the way computed answers are extracted from derivations in the IPP and in SLDNFA before discussing the additional operations which are not present in the CPP. It helps to understand the motivation behind some of those operations.

The IPP's definition of computed answer is similar to definition 3.3.2 of $T_e(A)$ (which is because the latter has been motivated by the former): Let $D$ be a disjunct of the current goal which does not contain false and to which the IPP operations have been exhaustively applied. Let $\sigma$ be any ground substitution such that all equalities and disequalities in $D$ are satisfied (true in CET). Then the conjunction of all non-equality atoms in $D\sigma$ is a computed answer derived by the IPP. For example, if the goal

$$p(X) \land X \neq a$$
is derived by the IPP, then \( p(t) \) is a computed answer where \( t \) is any ground term.

Note that all abducible atoms not present in a computed answer \( A \) are implicitly assumed to be false because \( \text{Comp}(A) \) is used in conditions (IPP2) and (IPP3). The fact that computed answers are not completed in the CALOG framework is an important difference between the CPP and the IPP. As a consequence, the CPP allows to distinguish between the case where a negative literal \( \neg A \) of an external predicate is explicitly derived (in the form of an implication \( A \rightarrow \text{false} \)) and the case where the corresponding positive atom \( A \) is simply absent from a computed answer (whereas these two cases are treated as identical by the IPP).

The following example illustrates the difference between the two approaches: whereas the IPP regards abducibles as undefined, the CPP assumes that abducibles do have definitions which are, however, inaccessible so that the CPP has to take into account all possible definitions in order to satisfy condition (A2) of the declarative semantics (and also condition (A3i), but only for ideal answers).

**Example 4.2.3** Consider the theory

\[ T_u: p \leftrightarrow \neg q \]

and the goal

\( p \)

The predicate \( q \) used in the definition of \( p \) is an abducible. After unfolding and a logical equivalence transformation

\( q \rightarrow \text{false} \)

is derived by both the CPP and the IPP, and the computation terminates. The IPP then discards the implication and returns \text{true} as a computed answer. This answer is correct in the IPP semantics, which assumes that \( q \) did not previously have a definition and can be defined as

\( q \leftrightarrow \text{false} \)

If this happens to be the inaccessible definition for \( q \) in the theory \( T_u \), then the
answer true is also correct with respect to the CALOG framework’s semantics, but this cannot be verified. Therefore, the CPP returns the implication \( q \rightarrow \text{false} \) as the computed answer so that, even if \( q \) is defined as true in \( T \), the crucial condition (A2) will trivially hold because the computed answer is then equivalent to false in \( T \) (and hence condition (A2) is rendered vacuous). Note that in this case the computed answer would not be an ideal answer, but it would be a good answer as condition (A3₂) only checks consistency with \( T' \).

As condition (SLDNFA1) of the semantics for SLDNFA indicates, SLDNFA does not have to return ground computed answers. A computed answer extracted by SLDNFA is the conjunction of the four components listed below. Rather than giving formal definitions, their relation to corresponding elements of a CPP derivation is described.

- **answer substitution**: corresponds to the substitution defined by the equality atoms in the final goal of a CPP derivation
- **set of abducible atoms**: corresponds to the set of suspended atoms (other than equalities) which also become part of an answer computed by the CPP
- **abductive completion**: if-and-only-if definitions for abducible atoms \( a(X) \) in the conditions of implications stating that \( a(X) \) holds exactly for the \( a \)-atoms which are elements of the set of abducible atoms (those variables corresponding to free variables in a CPP derivation are quantified existentially; see also definition 3.3.2 which prepares the setting for theorem 3.3.4)
- **negative abductive residue**: corresponds to the denials derived from the implications in the CPP when extracting answers (see section 3.1.2), except that no disequalities are added (since the information expressed by the disequalities is contained separately in the abductive completion) and
only implications with equalities in their conditions have to be considered (because again the others are covered by the abductive completion).

The CPP's definition 3.1.7 of computed answer simplifies the definition used by SLDNFA by merging the abductive completion and the negative abductive residue part of the SLDNFA answer. This also makes the answer more general as it leaves open the definitions of the abducibles. Thus inconsistencies with the actual definitions in $T_e$ may be avoided.

### 4.2.2.5 Generalized splitting

This section and the following two deal with the additional operations of the IPP and SLDNFA which are not in the CPP. The purpose of the operations is discussed and illustrated by examples. Actual or potential problems caused by the extra operations, especially with respect to (non-)minimality of computed answers, are also pointed out.

*Generalized splitting* is defined by the IPP as a logical equivalence transformation applied to implications:

$$(D \rightarrow A \lor Rest) \land C \leftrightarrow (A \land C) \lor ([D \rightarrow Rest] \land C)$$

where $A$ must not contain universally quantified variables and where $Rest$ may be empty (and hence equivalent to $\text{false}$ in the resulting implication $D \rightarrow Rest$). Note that the distributive splitting operation of the CPP,

$$(A \lor B) \land C \leftrightarrow (A \land C) \lor (B \land C)$$

is obtained as a special case of generalized splitting by setting $Rest \equiv \text{false}$ and $D \equiv \neg B$.

The restriction that $A$ must not contain universally quantified variables is to avoid a violation of the allowedness restriction — atomic subgoals must never contain universally quantified variables. This condition is always satisfied in the case of the distributive splitting operation (because $A$ and $B$ already were atomic subgoals before the splitting). In SLDNFA’s corresponding operation, which does not introduce a disjunction but gives rise to two
different derivations, the same effect is achieved by imposing the safe selection requirement.

The general splitting operation makes strong completeness results obtainable.

Example 4.2.4

\[ T_u : p \leftrightarrow \neg q \]
\[ q \leftrightarrow r \land \neg a \]
\[ r \leftrightarrow r \]
\[ G_0 : p \]

The CPP transforms the goal into the implication
\[ r \rightarrow a \]
and then goes into an unfolding loop and does not generate an answer. Instead, the IPP (and also SLDNFA, but using a different representation) splits the implication, obtaining the disjunction
\[ a \lor (r \rightarrow \text{false}) \]
While the IPP also does not terminate when unfolding \( r \) in the second disjunct, it does extract the answer \( a \) from the first disjunct.

Generalized splitting has several disadvantages. It is an expensive operation as it may considerably enlarge the search space. It also gives rise to the computation of non-minimal answers as in the following example.

Example 4.2.5 (Non-minimal answers due to generalized splitting)

Suppose that in example 4.2.4 the definition of \( q \) is replaced by
\[ q \leftarrow b \land \neg a \]
where \( b \) is undefined. Then the goal \( p \) is reduced to
\[ b \rightarrow a \]
and generalized splitting gives
\[ a \lor (b \rightarrow \text{false}) \]
Hence two computed answers are obtained — the unique minimal answer with
an empty set of abducibles from the second disjunct and the non-minimal answer a from the first disjunct.

[DeDS97] address the problem of (non-)minimality by proposing alternative notions of minimality and by suggesting that non-minimal answers could be identified and eliminated by examining all computed answers after they have been generated.

To avoid these problems, generalized splitting has not been included in the CPP, sacrificing stronger completeness results which would otherwise be obtainable. It could, of course, be added as an additional operation if required.

4.2.2.6 Factoring

The factoring operation of the IPP and of SLDNFA° (which is an extension of SLDNFA), replaces the conjunction

\[ p(X) \land p(Y) \]

where p is an abducible, by the disjunction

\[ [ p(X) \land p(Y) \land Y \neq X ] \lor [ p(X) \land Y = X ] \]

i.e. factoring distinguishes between the case where X and Y are equal and where they are distinct. Factoring can be understood as a shortcut for adding a tautology, splitting, equality rewriting and subsumption. In the example:

\[ p(X) \land p(Y) \land (Y = X \lor Y \neq X) \]

can be split, and yields, after equality rewriting:

\[ [ p(X) \land p(Y) \land Y \neq X ] \lor [ p(X) \land p(X) \land Y = X ] \]

One of the two p(X) atoms can then be removed by subsumption to yield the same result as factoring.

The following example shows the necessity of the factoring operation.

Example 4.2.6

\[ \mathcal{I}_u : p(Z) \leftrightarrow q(X, Z) \]
\[ \mathcal{IC} : q(Y, Z) \rightarrow p(Z) \]
$G_0 : p(a)$

After adding the integrity constraint to the initial goal, $p(a)$ can be unfolded, yielding

$$q(X, a) \land [ q(Y, Z) \rightarrow p(Z) ]$$

Now propagation can be applied:

$$q(X, a) \land p(a) \land [ q(Y) \rightarrow p(Z) ]$$

Now $p(a)$ can be unfolded again, yielding

$$q(X, a) \land q(X', a) \land [ q(Y, Z) \rightarrow p(Z) ]$$

These last two steps, propagation followed by unfolding, could be applied again and again, but then no computed answer will be obtained. Instead, factoring can be applied to the last goal and gives (after splitting)

$$\left( q(X, a) \land q(X', a) \land X' \neq X \land [ q(Y, Z) \rightarrow p(Z) ] \right) \lor$$

$$\left( q(X, a) \land X' = X \land [ q(Y, Z) \rightarrow p(Z) ] \right)$$

In the second disjunct propagation has already been exhaustively applied, so the IPP can extract the computed answers

$$q(t, a)$$

where $t$ is any ground term. The other disjunct gives rise to a non-terminating computation which yields, by further applications of the factoring operation, infinitely many additional non-minimal answers (such as $q(t_1, a) \land q(t_2, a)$ where $t_1$ and $t_2$ are two distinct ground terms).

Example 4.2.6 is based on a simpler example (without variables) given in [Fu96]. However, a close inspection of Fung's example reveals that (generalized) splitting of the integrity constraint could have been used instead of factoring; so Fung's example did not prove the necessity of factoring. Unfortunately, generalized splitting cannot be applied in example 4.2.6 because of the restriction that the conclusion of the implication to be split must not contain universally quantified variables (here $p(Z)$ contains the universally quantified variable $Z$).

Denecker and DeSchreye point out in [DeDS97] that factoring may be computationally expensive and propose to restrict the operation to a subset of the
abducible predicates explicitly identified by the user.

4.2.2.7 Case analysis for equality

The IPP and the SLDNFA extension SLDNFA+ define a special operation to deal with equalities in the conditions of implications. If $X$ is a free or existentially quantified variable, then

$$(A \land X = t) \rightarrow B$$

is rewritten to

$$X \neq t \lor [X = t' \land (A \rightarrow B)]$$

where $t'$ is $t$ with any universally quantified variables replaced by fresh existentially quantified variables.

Case analysis preserves logical equivalence — the operation can be simulated by introducing the tautology $X = t' \lor X \neq t$, (generalized) splitting and subsumption.

The IPP uses case analysis for equality in order to compute all minimal ground answers. The operation is also needed to guarantee correctness of computed answers with respect to the underlying semantics (conditions (IPP1)–(IPP3)). The following example illustrates both aspects.

Example 4.2.7 (adapted from [DeDS92, DeDS97, Fu96, FuKo96])

$$T_u : fault \leftrightarrow powerfailure(X) \land \neg backup(X)$$
$$backup(X) \leftrightarrow battery(X,Y) \land \neg empty(Y)$$
$$battery(X,Y) \leftrightarrow X=bat \land Y=cell$$

$$G_0 : fault$$

Then the steps in the derivation are (omitting logical equivalence transformations and equality rewriting):

$$powerfailure(X) \land \neg backup(X)$$
$$powerfailure(X) \land (battery(X,Y) \rightarrow empty(Y))$$

---

4This is a tautology: $\neg \forall x \neg p(x) \equiv \exists y p(y)$.
powerfailure(X) \land ( X=\text{bat} \rightarrow \text{empty(cell)} )

At this point the CPP and SLDNFA (without the extension) terminate and extract the computed answer

powerfailure(X) \land X \neq \text{bat}

where X is existentially quantified (as it is during the whole derivation).

If the IPP also terminated, it would discard the implication and extract the ground answers

powerfailure(t)

where t is any ground term, including t=\text{bat}. But since the IPP completes abducibles, \neg\text{empty(cell)} is implicit in all of these ground answers, which implies that powerfailure(\text{bat}) is (or would be) a wrong answer.

Therefore the IPP continues the derivation and applies case analysis to the equality in the condition of the implication, yielding (after splitting)

( powerfailure(X) \land X \neq \text{bat} ) \lor
( powerfailure(\text{bat}) \land \text{empty(cell)} \land X=\text{bat} )

The first disjunct yields the ground answers

powerfailure(t)

where t is any ground term not equal to \text{bat}. The second disjunct yields the additional ground answer

powerfailure(\text{bat}) \land \text{empty(cell)}

This answer is minimal according to the standard definition of minimality, but, as already mentioned in section 4.2.1, one might argue that powerfailure(t) for t \neq \text{bat} is "more minimal" in the sense that it does not require any additional assumptions (whereas powerfailure(\text{bat}) does). A simple way to formalize this notion of minimality is via minimal cardinality of the set of ground abducible atoms in the computed answer.

Just as SLDNFA has been extended to SLDNFA_+, the CPP could be extended to also incorporate case analysis, and ground computed answers could then be extracted in a similar way as by the IPP (but assignments to the free variables from the initial goal have to be kept in the form of explicit equality
atoms to satisfy condition (A2) of the semantic answer definition).

Alternatively, case analysis could be used only to simplify the computed answers generated by the CPP. In SLDNFA terminology, the negative abductive residue part of the computed answer (cf. section 4.2.2.4), which only contains implications with at least one equality atom in the condition, becomes empty after performing case analysis for equality exhaustively; thus fewer implications become part of the computed answer extracted by the CPP.

As has been mentioned above, case analysis can be simulated by the introduction of tautologies and the application of a sequence of equivalence preserving operations including generalized splitting. In fact, generalized splitting alone is sufficient to achieve most of the effects of case analysis, but it has (once again) the side effect of generating non-minimal answers. In example 4.2.7 generalized splitting applied to the implication on which case analysis was performed gives

\[
powerfailure(X) \land ( X\neq \text{bat} \lor \text{empty(cell)} )
\]

which can then be transformed by ordinary splitting to yield

\[
( powerfailure(X) \land X\neq \text{bat} ) \lor \\
( powerfailure(X) \land \text{empty(cell)} )
\]

The first disjunct is the same as after performing case analysis in example 4.2.7, but in the second disjunct the assignment \(X=\text{bat}\) has been lost. Thus the IPP generates the ground computed answers

\[
powerfailure(t) \land \text{empty(cell)}
\]

where \(t\) is any term. All of these answers, except for the one where \(t=\text{bat}\), are non-minimal. Case analysis can thus be regarded as a special case of generalized splitting which preserves minimality of answers (at least in the usual sense).

The case analysis operation could be generalized for abducible atoms, i.e. if an abducible atom appears in the condition of an implication, e.g.

\[
p \land a \rightarrow b
\]
then two cases could be considered — one in which the abducible atom \( a \) is true and one in which it is false:

\[-a \lor (a \land [p \rightarrow b])\]

As [Fu96] observes, this operation may lead to the generation of non-minimal answers because the case in which \( a \) is assumed to be true may not be required to explain the initial goal.

4.3 Constraint Logic Programming

4.3.1 The CLP(\( X \)) scheme

[JaLa87] introduced the general CLP(\( X \)) framework — Constraint Logic Programming over a given constraint domain \( X \).

4.3.1.1 Framework outline and semantics

Following [Ma93] and the survey [JaMa94], \( X \) may be regarded as a 4-tuple \((\Sigma, M, \mathcal{L}, \mathcal{D})\) where, somewhat informally

- \( \Sigma \) defines the available function and predicate symbols (and is called the signature)

- \( M \) specifies the domain and an interpretation for \( \Sigma \) (and is called the \( \Sigma \)-structure)

- \( \mathcal{L} \) is the set of constraints which can be expressed (usually a first-order language over \( \Sigma \) closed under conjunction and existential quantification)

- \( \mathcal{D} \) is a first-order \( \Sigma \)-theory (the domain theory) axiomatizing some properties of \( M \) (see below)

The domain theory \( \mathcal{D} \) and the model \( M \) are required to correspond on \( \mathcal{L} \): \( M \) has to be a model of \( \mathcal{D} \) and has to agree with \( \mathcal{D} \) on the satisfiability of constraints:
\( \mathcal{M} \models \exists C \iff \mathcal{D} \models \exists C \) for every constraint \( C \in \mathcal{L} \).

Furthermore, \( \mathcal{D} \) has to be satisfaction complete with respect to \( \mathcal{L} \): either
\[
\mathcal{D} \models \exists C \quad \text{or} \quad \mathcal{D} \models \neg \exists C
\]
for every constraint \( C \in \mathcal{L} \) (cf. definition 3.3.1).

[JaMa94] point out that it is possible and has been done in [Ma87] to start with a satisfaction complete domain theory \( \mathcal{D} \) and introduce the structure \( \mathcal{M} \) only afterwards (as the intended model of \( \mathcal{D} \)). While the resulting framework remains the same, the approach becomes conceptually closer to the CALOG framework: the theory \( \mathcal{D} \) corresponds to the theory \( \mathcal{T}_b \) of built-in predicates.

Example 4.3.1 Let \( \mathcal{M} \) be the domain of real numbers, with \( \Sigma \) providing addition and multiplication as function symbols and equality and inequality as predicate symbols, and \( \mathcal{M} \) interpreting them as usual. Let \( \mathcal{L} \) be the conjunctive and existential closure of the set of primitive constraints, which are simply all atoms which can be formed in \( \Sigma \), i.e. \( \mathcal{L} \) contains constraints of real arithmetics, e.g.
\[
\exists X, Y (2X = Y \land X + 1 \geq Y)
\]
Let \( \mathcal{D} \) be a first-order theory formalizing rules of arithmetic such that \( \mathcal{D} \) corresponds to \( \mathcal{M} \) and is satisfaction complete with respect to \( \mathcal{L} \) (it is not important what \( \mathcal{D} \) actually looks like; it is mainly used as a theoretical tool to formulate theorems). The resulting system is usually referred to as CLP(\( \mathcal{R} \)).

A built-in constraint solver, which is a black-box from the user's point of view (cf. section 4.3.2), is used to handle constraints. For example, a constraint solver for CLP(\( \mathcal{R} \)), i.e. for constraints of linear real arithmetic, usually has special-purpose algorithms for solving systems of linear equations and inequalities — like Gaussian elimination and some variant of the Simplex algorithm — built into it.

It is assumed that equality is available in \( \Sigma \), interpreted by \( \mathcal{M} \) as the identity relation and handled by a unification algorithm built into the constraint
solver. As has been frequently pointed out in the CLP literature (see especially [Ma93]), standard LP can thus be viewed as CLP(Herbrand), i.e. CLP over the Herbrand domain (regarded as the structure of finite trees constructed by the function symbols in $\Sigma$) with equality being the only constraint predicate and the unification algorithm being the constraint solver.

The user specifies a goal $G_0$ and a logic program $P$ with clauses of the form

$$H \leftarrow B \land C$$

where $H$ is an atom of a user-defined predicate, $B$ a conjunction of atoms of user-defined predicates and $C \in \mathcal{L}$ a constraint (i.e. usually a conjunction of literals of constraint predicates). So user-defined predicates can be defined in terms of constraint predicates, but they are not treated (by the constraint solver) as constraints themselves. The answer to $G_0$ is returned in the form of an answer constraint $C \in \mathcal{L}$ which has to satisfy the following conditions:

1. $C$ consists of atoms of constraint predicates only
2. $\mathcal{P} \cup \mathcal{D} \models \forall G_0 \leftarrow C$
3. $\mathcal{D} \cup \exists C$ is consistent

Condition (1) is the CLP instance of condition (A1) in the CALOG framework. Condition (2) corresponds to condition (A2), with $\mathcal{P}$ corresponding to $T_u$, $\mathcal{D}$ corresponding to $T_e$ and $T_\varnothing$ being empty. The satisfiability condition (3) corresponds to the good answer condition (A3g) (note that since $C$ consists only of atoms of constraint predicates, adding $\mathcal{P}$ on the left side in condition (3) makes no difference).

$\mathcal{D}$ is used in the semantics instead of $\mathcal{M}$ for a rather technical reason and it could be replaced by $\mathcal{M}$ if (and only if) $\mathcal{P}$ is canonical [JaLa87].

5A logic program $\mathcal{P}$ is canonical if its greatest fixpoint $GFP(\mathcal{P})$ is equal to the greatest lower bound of the standard fixpoint operator $T_\varnothing$, i.e. if $GFP(\mathcal{P}) = T_\varnothing \uparrow$. If this is not the case, then $\mathcal{P}$ has a non-empty infinite failure set which may make completeness and refutation soundness results unobtainable. See [JaLa87] and others for details.
4.3.1.2 Solving constraints

Operationally, the answer constraint \( C \) to a goal \( G_0 \) is computed by a derivation starting from \( G_0 \) and using the program \( P \) for unfolding atoms of user-defined predicates. Any constraints encountered in the process are stored in a separate constraint store on which the constraint solver operates. Initially, the constraint store is empty.

The constraint solver performs two operations called (in [JaMa94]) infer and consistent. The operation infer transforms a given constraint store \( C \) into a new constraint store \( C' \) which is equivalent to \( C \) and, if possible, simpler and more informative. Whether infer is able to simplify \( C \) depends on \( C \) and also on efficiency considerations on which the implementation of infer may be based (i.e. even if a simplification is possible in theory, it may be too inefficient in practice). The only formal requirement which infer has to satisfy is

\[
D \models \tilde{V}(C \leftrightarrow C')
\]

i.e. the two constraint stores have to be equivalent in the domain theory \( D \). This ensures soundness of the constraint solver with respect to condition (2) of the semantics. E.g. the expression from example 4.3.1

\[
\exists X, Y (2X = Y \land X + 1 \geq Y)
\]

may be transformed into

\[
\exists X, Y (2X = Y \land 1 \geq X)
\]

as the two conjunctions are equivalent in the domain of real arithmetic.

The second operation, consistent, checks whether

\[
D \models \exists C
\]

i.e. whether the constraint store is satisfiable in \( D \). This ensures soundness of the constraint solver with respect to condition (3) of the semantics. E.g.

\[
\exists X, Y (X < Y \land Y < X)
\]

is not satisfiable under the interpretation of \(<\) in real arithmetic.

Soundness and completeness results reported for CLP depend on the assumption that the operations infer and consistent are fully implemented as
sketched above. However, to make them and hence the constraint solver more efficient in practice, a full implementation of the constraint simplification in \texttt{infer} and of the satisfiability check in \texttt{consistent} is often avoided. To make the \texttt{infer} operation more efficient, constraints are separated into two classes, one of active constraints and one of passive constraints. The \texttt{infer} operation only works on ("solves") active constraints, whereas passive constraints are delayed until they become active, e.g. when variables become instantiated — if this never happens, then a constraint may remain passive throughout a derivation and simply be returned as part of the answer constraint. In the worst case, the answer constraint may be the unmodified conjunction of all constraints encountered in the derivation and it may not be satisfiable in \( D \).

4.3.2 Black-box, glass-box, no-box

The traditional approach to CLP, which has been formalized in the CLP\((X)\) scheme, is sometimes referred to as a black-box approach because, from the user's point of view, the constraint solving is done in a black-box, the built-in constraint solver. A major theoretical disadvantage of black-box approaches to CLP is that the constraint solver becomes part of the (operational) semantics: to fully understand the meaning of a computed answer (or lack thereof), it is necessary to know how the operations \texttt{infer} and \texttt{consistent} have been implemented (which is usually not possible if only because the detailed workings of the constraint solver are undisclosed for copyright reasons). Another practical disadvantage is that the techniques integrated into the constraint solver may not be the right ones to solve a given problem efficiently.

New approaches to CLP and extensions to the black-approach have been proposed to overcome these deficiencies. Glass-box constraint solving is understood as the user's ability to choose the method of constraint solving which is to be applied to a specific problem. Thus the user can guide the constraint solver in finding a solution more efficiently. For example, this is realized in the \texttt{cc(FD)} language [VHSaDe93] by several constraint operators (such as impli-
cation, propagation and cardinality) which can be employed as (higher-level) constraints in a program by the user. Similar possibilities exist in CHIP [VH89] where disjunctive constraints (i.e. constraints that are disjunctions; see especially section 5.1) can be labeled as forward-checking or look-ahead constraints. In black-box approaches these choices are all left to the constraint solver and the user is able only to specify the low-level constraints available in L.

The term glass-box constraint solver is sometimes also used to refer to a transparent constraint solver, i.e. a built-in solver whose rules are known to the user. The treatment of equality in the CALOG framework can be viewed in this way and the equality rewrite rules defined in section 3.1.1 effectively implement a transparent constraint solver for equality.

The no-box approach⁶ to CLP constitutes the most complete departure from the original black-box approach as it abandons the idea of a built-in constraint solver. Instead, the user specifies a set of Constraint Handling Rules (CHRIs) which implement a constraint solver explicitly. This approach is closest to the CALOG framework's and CHRIs are similar to integrity constraints (for more details see section 4.3.5).

It should be noted that black-box, glass-box and no-box approaches could be combined. For example, one could partition the set of built-in constraint predicates into two sets and have a black-box solver for the one set and a glass-box solver for the other. In addition, CHRIs (section 4.3.5) or integrity constraints could be used for user-defined constraints. Thus highly efficient general algorithms (such as the Simplex algorithm for solving linear programming problems) traditionally implemented by black-box solvers can be combined with more specialized approaches needed in certain situations which cannot be anticipated or easily recognized by a black-box solver. In the CALOG framework a transparent constraint solver is used for equality, and a no-box approach is taken to all other (constraint) predicates (note that any

⁶It appears that this fitting term has been coined by Frühwirth. In previous work, e.g. [We94], the no-box approach has also been referred to as a glass-box approach, mainly for lack of a better expression to contrast it with the black-box approach.
predicate can thus be treated as a constraint). However, a black-box solver for some or all other built-in predicates could be integrated into the framework as well.

Adding a black-box constraint solver may also be justified by the observation that a black-box solver could be understood as implementing a variant of theory resolution [EiOh93], a concept used in theorem-proving. For example, theory resolution for inequality is able to derive false from $x>1$ and $x>0 \rightarrow false$ — just like a black-box constraint solver.

### 4.3.3 Local propagation

Both to show the historical development of no-box constraint solving and to establish links between the CALOG framework and related CLP approaches, this section and the following two present increasingly powerful techniques for no-box constraint solving. The first is called local propagation [Le88].

In the terminology of the CALOG framework, the basic idea of local propagation is to have only integrity constraints whose condition is a single atom. This restriction allows a very efficient checking of integrity constraints, but it also limits the expressive power.

Local propagation looks at one constraint atom at a time and tries to derive information about the values of the variables occurring in the constraint. For example, the values of all three variables in the constraint $plus(X,Y,Z)$ are determined as soon as two of them are known. Local propagation rules in the form of single-condition integrity constraints can be used to make the necessary calculations:

\[
\begin{align*}
\text{plus}(X,Y,Z) & \rightarrow \text{minus}(Z,X,Y) \\
\text{plus}(X,Y,Z) & \rightarrow \text{minus}(Z,Y,X)
\end{align*}
\]

Here it is assumed that $plus$ and $minus$ atoms can be evaluated only if the first two parameters are ground — i.e. if they are not suspended according to the CALOG framework's definition of suspension. For example, if the constraint $plus(3,Y,5)$ is encountered, the first rule can be used to derive $\text{minus}(5,3,Y)$.
which can be evaluated to $Y=2$.

Unfortunately, local propagation already fails to solve simple systems of two equations with two unknowns, e.g.

$$X+Y=2 \land X-Y=0$$

can be solved only by propagation rules (or integrity constraints) with at least two condition atoms. Local propagation with the above (and similar) rules has no effect because two variables appear in both equations.

[Le88] discusses techniques to guess values, then apply local propagation, examine the error, make a better guess and hope for the guesses to eventually converge (this strategy is called relaxation).

A better option may be to restrict the application of local propagation to subproblems for which it is particularly suitable and allow steps of global propagation (which is simply any kind of propagation that looks at more than one constraint at a time) in between. This is possible in cc(FD) [VHSaDe93] where local propagation has been implemented as a higher-order constraint combinator, i.e. the user can specify implications

$$\text{Condition} \rightarrow \text{Conclusion}$$

to be used as local propagation rules whenever an atomic constraint matches the Condition.

In the CALOG framework, local propagation can be simulated by single-condition integrity constraints; see section 6.5.1 for an example application. Their application could be subjected to appropriate control mechanisms so that they are exclusively used if (and only if) it is efficient to do so.

4.3.4 Guarded rules

Before discussing Constraint Handling Rules for what might be called no-box global propagation in the next section, it is instructive to look at their historical predecessor, guarded rules, especially since one particular framework which uses guarded rules [Sm91] shares a number of interesting features with the CALOG framework.
The major motivation of adding guards to LP clauses is to reduce (or eliminate) don’t-know non-determinism: entailment of the guard part of a clause (by the constraint store) is tested before the clause is applied. This is particularly useful for concurrency and will be discussed in more detail in section 4.4.

While the syntax used for guarded rules differs, semantically a guarded rule is always an implication

\[ \text{Guard} \rightarrow \text{Clause} \]

where Guard is a conjunction of literals (or a constraint in CLP frameworks) and Clause can be any logical formula, but is usually either an implication or an equivalence.

Contrary to the standard usage of guarded rules as guarded program clauses (i.e. definitions), Smolka [Sm91] uses guarded rules in a way which is more similar to the CALOG framework’s integrity constraints to provide logically redundant, but operationally useful information on how to handle user-defined constraints. The Clause part of a guarded rule is an equivalence in [Sm91], and thus a guarded rule is of the form

\[ \text{Guard} \rightarrow (\text{Head} \leftrightarrow \text{Body}) \]

where Head is an atom and Body is a conjunction of atoms. If Head appears in the goal and the constraints in Guard can be shown to hold using the current constraint store (i.e. \( D \models (C \rightarrow \exists \text{Guard}) \), where \( D \) is the domain theory and \( C \) the constraint store), then Head may be replaced by Body in the goal.

Smolka requires guarded rules to be admissible in the sense that they have to be true in every model of the if-and-only-if definitions which constitute the actual program and which have the same form as definitions in the CALOG framework. So, in CALOG terminology, guarded rules have to be properties of definitions.\(^7\)

\(^7\)Smolka also discusses non-admissible guarded rules. While they need not cause problems in practice (operationally), there is no declarative semantics for them. The use of integrity constraints in the CALOG framework which are in a similar sense “non-admissible” (or non-logical) are discussed in the context of the deletion operation in chapter 5.
Example 4.3.2 (from [Sm91]) Consider the definition of list concatenation
\[
\text{app}(X,Y,Z) \leftrightarrow (X=[] \land Y=Z) \lor
(X=[H|T] \land Z=[H|U] \land \text{app}(T,Y,U))
\]
Then the following formulas are properties of the definition and thus admissible guarded rules:

\[
\begin{align*}
Y=Z & \rightarrow (\text{app}(X,Y,Z) \leftrightarrow X=[]) \\
Y=[] & \rightarrow (\text{app}(X,Y,Z) \leftrightarrow X=Z)
\end{align*}
\]

In the CALOG framework the following two integrity constraints approximate the meaning of the two guarded rules:

\[
\begin{align*}
\text{app}(X,Y,Y) & \rightarrow X=[] \\
\text{app}(X,\[],Z) & \rightarrow X=Z
\end{align*}
\]

Note that in this case the guards are made redundant by unification (i.e. the first integrity constraint has been obtained by simplifying \(\text{app}(X,Y,Z) \land Y=Z \rightarrow X=[]\)) whereas in general they have to become part of the conditions of the integrity constraints.

When used with the CALOG proof procedure, the integrity constraints yield similar effects as the guarded rules in Smolka's framework. For example, if \(\text{app}(X,Y,Y)\) is part of the goal, propagation with the first integrity constraint gives \(X=[]\), which can be used to obtain \(\text{app}(\[],Y,Y)\) by equality rewriting. This atom can then be unfolded to \(\[]=[] \land Y=Y\) which is removed by equality rewriting, leaving \(X=[]\) just as in the case of the guarded rules. In general, it may be necessary to remove some information explicitly by means of deletion rules (see section 5.4) in order to achieve the effect of guarded equivalences.

There is another important similarity between Smolka's approach and the CALOG framework. Smolka proposes the concept of residuation to further reduce search non-determinism. According to the residuation strategy, a goal atom has to be determinate before it can be reduced, which means that it must be possible to determine which of the disjuncts of an if-and-only-if definition applies. How this is determined, is left unspecified (i.e. it is done with the
help of a black-box constraint solver when necessary). The idea is similar to the CALOG framework's concept of suspension. E.g. in example 4.3.2, \texttt{app(X,Y,Y)} is not determinate as it could be reduced using both disjuncts of the definition of \texttt{app}. In the CALOG framework, the definition of \texttt{app} has to be rewritten into

\[
\texttt{app([],Y,Z) } \leftrightarrow \texttt{ Y=Z} \\
\texttt{app([H|T],Y,Z) } \leftrightarrow \texttt{ Z=[H|U] } \land \texttt{ app(T,Y,U)}
\]

so that \texttt{app(X,Y,Y)} is suspended. Then propagation and equality rewriting have the effect of "unsuspending" the suspended atom \texttt{app(X,Y,Y)} as they replace it by \texttt{app([],Y,Y)}, which only matches the first definition and is thus reducible.

Smolka proposes an additional means of control to unsuspend one atom in a goal all of whose atoms are suspended: certain predicates may be declared as \textit{generating}. Non-deterministic search over the definitions of atoms of those predicates is applied if all other atoms are suspended. Note that guarded rules may always be applied (provided that the guards are entailed), and an atom for which there is an applicable guarded rule is also called determinate (and is thus not suspended).

### 4.3.5 Constraint Handling Rules

\textit{Constraint Handling Rules} (CHR\textsc{s}) were introduced by Frühwirth in [Fr92, Fr95]) to provide explicit constraint handling for \textit{user-defined constraints} (any user-defined predicate can be regarded as a user-defined constraint). There are two main types of CHRs: simplification CHRs and propagation CHRs. Simplification CHRs are similar to Smolka's guarded rules (see previous section), but allow more than one atom in the \textit{Head} of a rule.\footnote{In practice CHRs allow only two head atoms. Note that a rule with two head atoms cannot be rewritten into rules with single-atom heads, but rules with more than two head atoms can be transformed into several rules with two head atoms.} They are written in the following way:
\[ \text{Head} \iff \text{Guard} \land \text{Body} \]

where \text{Guard} is a built-in constraint, \text{Head} is a conjunction of user-defined constraint atoms and \text{Body} is a conjunction of built-in and user-defined constraints. The semantics of a simplification CHR is the same as of Smolka's guarded rules:

\[ \text{Guard} \rightarrow (\text{Head} \iff \text{Body}) \]

In a propagation CHR the equivalence is replaced by a simple implication:

\[ \text{Head} \Rightarrow \text{Guard} \land \text{Body} \]

meaning

\[ \text{Guard} \rightarrow (\text{Head} \rightarrow \text{Body}) \]

which is logically equivalent to

\[ \text{Guard} \land \text{Head} \rightarrow \text{Body} \]

So declaratively, propagation CHRs are identical to integrity constraints in the CALOG framework, except that disjunctions are not allowed in their conclusions (i.e. in \text{Body}). Operationally there is a difference as some of the condition atoms are identified as guards (see section 5.2.5 for a discussion of extending the CALOG framework accordingly).

The effect of a simplification CHR (and hence also of the guarded rules introduced in the previous section) can be achieved by a propagation CHR (with the same \text{Guard}, \text{Head}, and \text{Body}) followed by a deletion of the head atoms from the goal. In fact the deletion is not really necessary, although it may significantly improve efficiency. Consequently, integrity constraints as used in the CALOG framework are conceptually as powerful as CHRs (which themselves are more powerful than the single-headed guarded rules presented in the previous section); the issue of identifying special rules for deletion is discussed in section 5.4.

In pointing out the differences between his CHRs and Smolka's guarded rules, Frühwirth remarks that guarded rules "are only used as 'shortcuts' (lemmas) for predicates, not as definitions for user-written constraints" [Fr95, p. 104]. This reflects the fact that predicates "defined" by CHRs do not need
to have definitions (although they may). Similarly, in the CALOG framework integrity constraints provide the only (accessible) information about external predicates, but they may also contain atoms of user-defined and built-in predicates, which have accessible definitions.

As simplification CHRs can also be regarded as rewrite rules for user-defined constraints and as several CHRs may be applicable to the same goal atom (or conjunction of goal atoms) at the same time, the question of confluence arises. A CHR program is confluent if a computation reaches the same final state regardless of the order in which CHRs are applied. Criteria for confluence of CHR programs are given in [AbFrMe96].

Since CHRs can be regarded as the CLP instance of the CALOG framework, soundness and completeness results obtained for CHRs may be transferable to a CLP specialization (and, if necessary, extension) of the general CALOG proof procedure. [AbFrMe96] presents soundness and completeness results for CHRs which are claimed to be stronger than the ones usually obtained for CLP (cf. the semantics specified for CLP(X) in section 4.3.1.1). If \( \mathcal{P} \) denotes the CHR program and \( \mathcal{D} \) the domain theory defining equality and the (other) built-in constraints (which are used to evaluate the guards of CHRs), then

\[
\mathcal{P} \cup \mathcal{D} \models \hat{V}(G_0 \leftarrow C)
\]

where \( G_0 \) is the initial goal and \( C \) a computed answer constraint. Compare this to condition (2) of the CLP(X) semantics — what was a simple implication there \( (G_0 \leftarrow C) \), is now an equivalence.

Moreover, if \( \mathcal{P} \cup \mathcal{D} \models \hat{V}(G_0 \leftrightarrow C) \) and \( C \) is satisfiable in \( \mathcal{D} \), then there is a successful computation starting from \( G_0 \) and terminating in a computed answer constraint \( C' \) such that

\[
\mathcal{P} \cup \mathcal{D} \models \hat{V}(C \leftrightarrow C')
\]

Again, this result is supposedly stronger than affirmation completeness results obtainable for CLP — in CLP(X) instances, only a disjunction of answer constraints \( C_1, \ldots, C_n \) can be derived, not a single answer constraint, and only

139
a simple implication, not an equivalence, can be shown to hold [JaMa94]:

\[ D \models (C \rightarrow \forall_{i=1}^{n} \exists C_i) \]

However, it should be noted that [JaMa94] presents conditions under which the implication becomes an equivalence and/or the disjunction collapses to a single answer constraint \( n = 1 \), and these conditions arguably still allow for at least as large a class of programs as CHR programs do. Moreover, there appears to be a crucial difference between the two results which the authors of [AbFrMe96] may have missed: in the result for CLP(\( X \)) only the domain theory \( D \) appears, not the program \( P \). The difference this makes is that the result of [AbFrMe96] allows a trivial proof procedure which simply returns the initial goal as a computed answer constraint since \( C' \equiv G_0 \) satisfies the condition imposed on \( C' \) by assumption. Maybe this could be fixed by removing \( P \) from the left side of the logical implication.

As has already been said, a major motivation of CHRs was to have a means of explicitly specifying a constraint solver for user-defined constraints (the no-box approach). In theory, CHRs can also be specified for the usually built-in constraints such as arithmetic inequality. However, for efficiency purposes CHRs may be combined with a built-in constraint solver for these predicates. This approach has been formalized in [Fr94, Fr95] as the CLP+CH framework (CLP with added rules for constraint handling). CHRs have been implemented as an extension to the Eclipse programming system (see [Fr95] for references) which itself is a CLP extension of PROLOG. Thus the extended Eclipse system may be seen as a realization of the CLP+CH framework.

### 4.4 Concurrency

The following two sections briefly discuss how and to what extent it may be possible to also incorporate concurrent LP and concurrent CLP into the CA-LOG framework. They are mainly meant as an indication of possible directions of future research.
4.4.1 Concurrent Logic Programming

A good account of concurrent LP languages can be found in Shapiro's survey [Sh89] on which this section is mainly based. The book [Ja91] is also a useful source for concepts of concurrent LP; it describes a concurrent LP language called *Conclog*.

Shapiro identifies several important differences between standard LP languages (such as PROLOG) and concurrent LP languages. These differences have to be taken into account when trying to incorporate concurrency into the CALOG framework.

The first distinction mentioned in [Sh89] is that standard LP languages are *transformational*, i.e. they receive the initial goal as their input and generate answers as their output, whereas concurrent LP languages are *reactive*, i.e. they constitute an open system whose purpose is to "maintain some interaction with the environment" [Sh89, 2.1.1]. This distinction is similar to the one between rational and reactive agents which [KoSa96] seeks to reconcile, mainly by adding integrity constraints to the rational agent to enable it to also behave in a reactive way. Integrity constraints in the CALOG framework can be interpreted in a similar way (the work [KoSa96] is related to this thesis).

The next difference between standard and concurrent LP which [Sh89] mentions is the type of non-determinism. Standard LP uses *don't-know non-determinism* which requires backtracking over wrong choices (and even over right choices, if all solutions are to be found), whereas concurrent LP relies on *don't-care non-determinism* (both of these expressions were introduced in [Ko79b]) which commits to whichever choice has been made. LP languages which exhibit don't-care non-determinism are therefore also called *committed-choice* languages. A commonly used way of achieving don't-care non-determinism is to suspend any choice until only one alternative is left. The CALOG framework already does this by requiring the heads of if-and-only-if definitions to be mutually exclusive and by suspending goal atoms until they are an in-
stance of the head of a definition (which is then guaranteed to be unique). However, disjunctions are allowed in the bodies of definitions and splitting the disjunction corresponds to initiating a non-deterministic search. The CALOG framework allows the user to control the amount of such non-determinism by writing definitions in different ways (the less homogenized the less don't-know non-determinism, cf. section 2.2). If no disjunctions are used in the bodies of definitions, then the behaviour of the proof procedure is not very different from that of a committed-choice language and no non-deterministic search takes place.

The definition of suspension used in [Sh89] is basically the same as in the CALOG framework — goal atoms are reducible only if they are instances of a definition specified by a program clause. [Ja91] introduces some additional flexibility into the concept of suspension by allowing explicit user-defined suspension of the form

\[ \text{suspend } \text{Atom until Condition} \]

\[ \text{e.g.} \]

\[ \text{suspend append(X,Y,Z) until ground(X) } \land \text{ ground(Y)} \]

A similar mechanism is made available by the suspend predicate of the Eclipse programming system (cf. the section on CHRs above and see [Fr95] for references).

Program clauses in concurrent LP languages usually take the form of guarded rules (cf. section 4.3.4)

\[ \text{Head } \leftarrow \text{Guard | Body} \]

If a goal atom is an instance of Head and the condition(s) expressed in Guard can be shown to hold, then Body is derived as a new goal. In many concurrent LP languages (called flat languages) the atoms occurring in the Guard are restricted to a special class of guard predicates such as equality, inequality and arithmetic operators — basically the built-in predicates in the CALOG

---

\[ ^9 \text{Note that another way of achieving don't-care non-determinism is to use mutually exclusive guards in clauses whose heads unify.} \]
framework. This restriction is similar to the restriction imposed on guards in CLP that they have to consist of atoms of constraint predicates only.

In line with their classification as reactive, only an operational semantics is specified for concurrent LP languages in [Sh89]. A computation is successful if the goal is reduced to true. If a computation leads to a state in which all goal atoms are suspended, the computation is deadlocked. Equality is treated just as in standard LP, i.e. equality atoms are not represented explicitly but substitutions are stored along with the goals. In the CALOG framework suspended equality atoms become part of the answer instead, and the distinction between successful and deadlocked computations is removed as answers are defined as consisting of suspended atoms.

The concept of concurrency is introduced by using a proof procedure which works on several or all unsuspended (reducible) goal atoms in parallel. An atom is then called a process and processes communicate by instantiating shared variables. They may synchronize simply by waiting for semaphore (signal) variables to become instantiated. [Sh89] emphasizes that LP frameworks are thus specially suited for implementing concurrency. Guards constitute another means of achieving synchronization (a process has to wait until the guard of one of its rules becomes true).

To conclude (for now), the CALOG framework certainly shows potential for incorporating many of the features of concurrent LP languages — integrity constraints provide for reactivity, and if-and-only-if definitions without disjunctions in their bodies together with the concept of suspension yield don’t-care non-determinism. It appears possible to make the operations of the CALOG proof procedure concurrent for a class of theories restricted to such definitions, but the details shall be left for future research.

4.4.2 Concurrent Constraint Logic Programming

As has been pointed out in the previous section, one characteristic feature of concurrent LP languages is don’t-care non-determinism or, in other words, the
fact that they are committed-choice languages. Maher first introduced a class of committed-choice constraint LP languages called ALPS in [Ma87]. Just like a concurrent LP program, a program in ALPS is a collection of guarded rules, but now the guards consist of atoms of constraint predicates, a requirement analogous to the one of flat concurrent LP languages. The constraints in the guard have to be entailed by the constraint store before a rule can apply.

Saraswat extended this concept and formalized a unified approach to Concurrent and Constraint LP (CCLP) in his Ph.D. thesis (see [SaRi90] and the book [Sa93]). Basically, Saraswat split the guard part of a guarded rule into two parts called ask and tell so that a CCLP program consists of clauses which may be written as

\[ \text{Head} \leftarrow \text{ask} : \text{tell} \mid \text{Body} \]

This syntax, which is used in [JaMa94], is not standard in CCLP languages, but makes a comparison with both concurrent and constraint LP languages easier. In order to be applicable, the Head of a program clause has to match some atom in the goal (matching can, but need not, be defined in the same way as in many concurrent LP languages and the CALOG framework, i.e. that a goal atom has to be an instance of Head), and certain semantic conditions imposed on ask and tell must be satisfied.

The semantics of the ask part corresponds to that of the concurrent LP guards, except that, as in Maher’s ALPS languages, the guards have to be constraints and it is checked whether they are entailed by the constraint store. Operationally, the ask constraints can be used to synchronize concurrently executing agents: the agents are delayed until the ask part of the guard is entailed.

The tell part also consists of constraints, but they only have to be consistent with the constraint store to which they are then added. Hence the terminology: an agent “asks” whether certain constraints on which it depends hold and “tells” the constraint store which other constraints the agent implies. These constraints can then trigger other agents which “ask” for the constraints.
Several CCLP languages have been developed (see [Sa93]) and many extensions have been proposed, e.g. the language Oz [Sm95] is now presented as an object-oriented CCLP language [HeSmWü95]. Oz also features a higher-order combinator for encapsulated search [ScSm94] which is discussed in section 5.1.4.6 where it is compared to the CPD methods available for similar purposes in the CALOG framework.

Similar remarks as in the previous section apply as far as integrating concurrent CLP into the CALOG framework is concerned — the framework may be compatible, but more work is required, especially on extending the proof procedure to actually exhibit concurrent behaviour.
Chapter 5

Extensions

This chapter discusses several ways of making the proof procedure defined for the CALOG framework in chapter 3 more efficient and more powerful, especially the propagation operation. Most notably, the chapter defines the CPD techniques for propagating across disjunct boundaries (section 5.1) which can be used to achieve a considerable pruning of the search space in several applications. Other interesting extensions of the proof procedure are the subsumption operation (section 5.3) and the deletion operation (section 5.4).

Possible extensions of the proof procedure by other operations — generalized splitting, factoring and case analysis — have already been discussed in chapter 4 when comparing the CALOG proof procedure with the Iff Proof Procedure and SLDNFA (see sections 4.2.2.5–4.2.2.7).

5.1 Constraint Propagation with Disjunctions (CPD)

In CLP, a disjunctive constraint is a disjunction of constraints. Since any predicate can be treated as a constraint in the CALOG framework, any disjunction may be regarded as a disjunctive constraint. New methods for handling such disjunctive constraints were proposed informally in [Ko92b] and a first formalization can be found in the author’s M.Sc. thesis [We94]. The methods are now called Constraint Propagation with Disjunctions (CPD).
This section first introduces the different CPD methods informally, motivating them individually by means of small examples. The formal definition follows, and then another example is given to show the power of the techniques in context. Finally, the CPD methods are compared with several standard CLP techniques for handling disjunctive constraints.

5.1.1 Motivation

5.1.1.1 \( \mathcal{NP} \) problems

Disjunctive constraints are commonly used in CLP to represent problems of Operations Research (see also section 6.3) such as various types of scheduling applications. In a job-shop scheduling problem, the starting times, \( t_1 \) and \( t_2 \), of two tasks which have to be processed on the same machine are constrained by a disjunction

\[
(t_1 + p_1 \leq t_2) \lor (t_2 + p_2 \leq t_1)
\]

where \( p_1 \) and \( p_2 \) are the respective processing times of the tasks. A solution to a job-shop scheduling problem is a schedule (i.e. an assignment to the starting times of the tasks) which satisfies the constraints (the disjunctive ones and some additional constraints, see section 6.3.1).

If \( n \) tasks have to be scheduled on one machine, then there are \( n \cdot (n-1)/2 \) disjunctive constraints for this machine. Since the disjunctions are all binary, this yields a search space with \( 2^{n \cdot (n-1)/2} \) nodes (again, for one machine only). The exponential growth of the search tree suggests that the problem is in the class of \( \mathcal{NP} \) problems, and it is indeed \( \mathcal{NP} \)-complete.\(^1\)

Suppose that five tasks are to be scheduled on the same machine so that

\[^1\text{A problem belongs to the class of non-deterministic polynomial or } \mathcal{NP} \text{ problems if it is solvable in polynomial time on a non-deterministic Turing machine [GaJo79]. This means that any path in the search tree is of at most polynomial length (which implies that a deterministic machine can verify or falsify any potential solution to the problem in polynomial time). A problem is } \mathcal{NP} \text{-complete if every other } \mathcal{NP} \text{ problem can be reduced to it in polynomial time. Hence, if an } \mathcal{NP} \text{-complete problem can be solved in polynomial time, then the same is true for all } \mathcal{NP} \text{-problems (which appears to be highly unlikely, but has not yet been disproved).} \]
there are ten disjunctive constraints of the above type. It could happen that by splitting the tenth disjunction first, only one of the two choices remains possible in the ninth so that after splitting it the other can be rejected immediately. Deciding the ninth may similarly reject one of the two choices in the eighth and so on. In the optimal case, a path in the search tree leading directly to the optimal solution is found by splitting only 10 times and examining only 20 nodes of the search tree.

However, starting by splitting the first constraint may not help at all in deciding any of the others and, in the worst case, the search tree may have to be completely expanded all the way down to the tenth choice before any nodes can be rejected. In this case all $2^{10} = 1,024$ nodes of the search tree have to be examined and the same operations of the proof procedure may have to be applied again and again in many different nodes. For example, the last disjunction is split 512 times and the same propagation operations may be triggered in many of the resulting 1,024 nodes.

This discussion illustrates two points. Firstly, the order in which disjunctions are split matters. In the Operations Research literature the problem of pruning the search tree is usually approached by proposing heuristic algorithms to decide which disjunction to split first at any stage in the search.

Secondly, while such heuristics can always prove useful, it would be even better if splitting could be avoided altogether or at least delayed as long as possible. This can be achieved by using all available information to decide disjunctions before splitting them and also by extracting additional information from some disjunctions which may help in deciding others. The necessary operations can be computationally expensive, but they may pay off nevertheless because they may help to avoid doing the same work repeatedly in different nodes of the search tree. These are the main ideas behind the CPD methods which are now individually motivated.
5.1.1.2 Propagating into disjuncts

Suppose the starting times $t_1$ and $t_2$ of two tasks in a scheduling application are already known to have the lower and upper bounds represented by the following constraints:

\[ 0 \leq t_1 \leq 18 \land 5 \leq t_2 \leq 20 \]

Suppose the processing times of the tasks are $p_1 = 8$ and $p_2 = 15$, so the disjunctive constraint between the two starting times is

\[ (t_1 + 8 \leq t_2) \lor (t_2 + 15 \leq t_1) \]

Now, if $t_1 + 8 \leq t_2$ holds in the first disjunct and $t_2 \leq 20$ holds "globally", then $t_1 \leq 12$ is implied in the first disjunct. Also, since $0 \leq t_1$ holds globally, it follows that $8 \leq t_2$. So these two constraints could be propagated into the first disjunct and, by a similar argument,

\[ t_2 \leq 3 \land 20 \leq t_1 \]

could be propagated into the second disjunct of the disjunctive constraint.

The information added to the second disjunct is inconsistent with the constraints on the starting times which were already known globally: $t_2$ has to be greater than 5 outside the disjunct and smaller than 3 inside the disjunct; $t_1$ has to be both greater than 20 and smaller than 18. Thus $false$ could be propagated into the disjunct.

Thus the second disjunct can be eliminated, deciding the disjunctive constraint so that the first disjunct can be added conjunctively as a global constraint to the goal — together with the stronger constraints on the starting times already propagated into it.

Note that even if a disjunction cannot be decided by propagating constraints into it, the derivation of tighter bounds within the disjuncts need not have been in vain: when the disjunction is split, the stronger constraints can immediately be used for further propagation.

Since the information outside a disjunction can be regarded as "global" information as opposed to the information which is "local" to each one of its
disjuncts, propagating into a disjunct will also be referred to as *global-to-local* propagation.

### 5.1.1.3 Propagating within disjuncts

As mentioned above, one should not expect that propagating into disjuncts is always able to decide a disjunction by propagating `false` into it and thus eliminating all but one disjunct.

Using again bounds on starting times as an example, it might be that the (local) bounds do get tighter by propagating global information into disjuncts, but not tight enough to yield a contradiction with other (global) constraints outside the disjunction.

However, in some cases accumulating constraints within disjuncts may make it possible to derive a contradiction within the disjunct. For example,

\[ t_1 \leq 10 \]

may be propagated into a disjunct which already contains

\[ 15 \leq t_1 \]

Then the denial integrity constraint

\[ X \leq Y \land Y \leq Z \land X > Z \rightarrow false \]

could be used to derive `false`, but this time by propagation within the disjunct. Note that this type of propagation is identical to ordinary propagation in the proof procedure, except that it does not require rewriting the goal into disjunctive normal form before it can be applied.

As mentioned in the previous subsection, information within a disjunct may be regarded as local information (as opposed to the global information outside the disjunction) and propagation within a disjunct will thus also be referred to as *local-to-local* propagation.
5.1.1.4 Propagating out of disjuncts

Sometimes it is not possible to decide a disjunctive constraint by propagating into it or within it. Still, it may be possible to extract useful information out of the disjunction before splitting it: local constraints which hold in all disjuncts may be made global.

Such constraints may be represented explicitly, e.g.

\[(t_1 \leq 4 \land 0 \leq t_2) \lor (t_1 \leq 4 \land t_2 \leq 10)\]

allows \( t_1 \leq 4 \) to be propagated out of the disjunction by simply applying the distributivity law. Or they may be implicit, e.g.

\[t_1 \leq 5 \lor 10 \leq t_1\]

implies that \( t_1 \) cannot take any value in the open interval \((5, 10)\). This information can be represented by the constraint \( t \not\in (5, 10) \) which could be propagated out of the disjuncts and could prove useful for further propagation.

As propagation out of disjuncts makes local information global, the method will also be called *local-to-global* propagation.

To employ a uniform terminology for all types of propagation, ordinary propagation among global information will also be referred to as *global-to-global* propagation.

5.1.2 Formalization

Before formalizing the CPD methods, some simplifying assumptions will be made regarding the structure of goals appearing in a derivation.

**Definition 5.1.1** A conjunction \( G \) is called a *simple goal* if it is of the following form:

\[ G = C_1 \land \ldots \land C_n \land D_1 \land \ldots \land D_m \]

where \( n, m \geq 0 \) and \( n + m > 0 \), where the \( C_i, 1 \leq i \leq n \), are literals or implications, called *global constraints* within \( G \), and where the \( D_i, 1 \leq i \leq m \), are disjunctions
$D_i = D_{i1} \lor \ldots \lor D_{i,k_i}$

with each disjunct $D_{ij}$, $1 \leq j \leq k_i$ being a conjunction of atoms and implications:

$D_{ij} = L_1 \land \ldots \land L_r$

The $L_k$, $1 \leq k \leq r$ are called local constraints within $D_{ij}$.

Note that both global and local "constraints" can be atoms of user-defined, built-in or external predicates.

According to definition 2.2.1, every initial goal is a simple goal. It will now be assumed that any further goals generated during a derivation are disjunctions of one or more simple goals.

It is easy to refine the proof procedure to ensure that this is the case. First observe that propagation and equality rewriting always transform a simple goal into another simple goal. If a goal is a disjunction of simple goals, then splitting can only be applied within a simple goal and as a result the simple goal is replaced by a disjunction of two simple goals, thus preserving the structure of the goal. Other logical equivalence transformations cause no problems either. The only operation which has to be restricted is unfolding: if atoms within a disjunction $D_i$ inside a simple goal are unfolded, then the result may not be a simple goal, and this is the case if and only if such unfolding introduces a disjunction. So this must be prohibited. Note that this does not affect any soundness or completeness results as the prohibition only delays unfolding, but it will eventually become applied (after splitting the disjunction).

It will be assumed in the remainder of this section that every goal in a derivation is a disjunction of simple goals. If a goal is a disjunction of two or more simple goals, then operations of the proof procedure can be applicable only within one of the simple goals. Therefore it suffices to consider simple goals in the following.

It is now straightforward to formalize the CPD methods for simple goals.
Definition 5.1.2 (CPD)

Suppose that
\[ G = C_1 \land \ldots \land C_n \land D_1 \land \ldots \land D_m \]
is a simple goal as in definition 5.1.1 and suppose that
\[ D_{ij} = L_1 \land \ldots \land L_r \]
is a disjunct of the disjunction \( D_i \). Then

- **Global-to-global CPD** is propagation between two global constraints \( C_s \) and \( C_t \). The resolvent is added as a conjunct to \( G \).

- **Global-to-local CPD** is propagation between a global constraint \( C_s \) and a local constraint \( L_t \). The resolvent is added as a conjunct to \( D_{ij} \).

- **Local-to-local CPD** is propagation between two local constraints \( L_s \) and \( L_t \). The resolvent is added as a conjunct to \( D_{ij} \).

Finally, assume that all disjuncts \( D_{ij} \) of the disjunction \( D_i \) contain the same local constraint \( L_s \). Then adding \( L_s \) as a conjunct to \( G \) is called (simple) **local-to-global CPD**.

It is easy to verify that the four CPD methods do not affect any of the soundness and completeness results obtained in chapter 3.

Global-to-global CPD and local-to-local CPD are just new names for special ways of applying ordinary propagation. Formalizing them in this new way facilitates the definition of new control mechanisms for the proof procedure: unrestricted propagation in all parts of a (non-simple) goal could be very inefficient, and restricting propagation to simple goals may yield computationally better results. As some preliminary experiments suggest (see chapter 6), prohibiting local-to-local propagation may often be advisable. An implementation could provide the user with appropriate control mechanisms (such as run-time flags for an interpreter or compiler switches for a compiler) to make such choices.
Global-to-local CPD is logically equivalent to the application of several steps of ordinary propagation to the disjunctive normal form of the goal.

Local-to-global CPD in its simple form is just a reversal of splitting, i.e., the logical equivalence transformation

\[(A \lor B) \land C \leftrightarrow (A \land C) \lor (B \land C)\]

is applied from right to left (but without removing C from the disjuncts). However, since the application of local-to-global CPD is restricted to disjunctions in simple goals, it will never undo an application of splitting by the proof procedure.

Local-to-global CPD can be made more powerful (and has to in order to be efficient in practice) by allowing it to derive the constraints common to all disjuncts rather than relying on them being present (as illustrated in the following).

In the second example given in section 5.1.1.4 the information common to the two disjuncts in the disjunction

\[t_1 \leq 5 \lor 10 \leq t_1\]

was

\[t \notin (5,10)\]

One way to program the identification of such implicit common information is by permitting disjunctive conditions in integrity constraints, e.g.

\[(x \leq y \lor z \leq x) \land z > y \rightarrow x \notin (y,z)\]

could be used for local-to-global CPD in the given example. As long as such integrity constraints are used only for local-to-global CPD, no further adaptations to the framework are required. Like any other integrity constraints, they should be properties of the theory \(T\) and this guarantees the soundness of local-to-global CPD (in the sense of lemma 3.3.1, i.e. a goal obtained by applying one step of local-to-global CPD is equivalent, in \(T' \cup CET\), to the previous goal).

In theorem-proving terminology, CPD can be understood as a form of non-clausal resolution [Mu82]. It may thus be worth exploring whether it is possible...
to apply known results from the theorem-proving literature to CPD.

5.1.3 4-queens example

This section shows how the four CPD methods can be applied in the n-queens problem (for \( n = 4 \)). Further applications of CPD are discussed in the applications chapter 6.

Recall from example 2.1.1 that a single integrity constraint is sufficient to solve the n-queens problem:

\[
\text{queen}(A, B) \land \text{queen}(C, D) \land \text{move}((A, B), (C, D)) \rightarrow \text{false}
\]

where \text{move} is defined to represent valid moves on an \( n \times n \) chessboard.

However, it is easier to illustrate the CPD methods if the following two integrity constraints are used instead:

\[
\text{mark}: \quad \text{queen}(A, B) \land \text{move}((A, B), (C, D)) \rightarrow \text{attacked}(C, D)
\]
\[
\text{dead}: \quad \text{queen}(C, D) \land \text{attacked}(C, D) \rightarrow \text{false}
\]

These two integrity constraints, which have been labeled for easier reference, make use of an additional external predicate \text{attacked} whose use, intuitively, is to store the coordinates already attacked by placed queens. Note that resolving the two integrity constraints with each other yields the integrity constraint from example 2.1.1.

The definition of \text{move} will be written in such a way that \text{move}((A, B), (C, D)) is reducible if and only if \( A \) and \( B \) are ground (cf. example 2.1.1, but see also example 5.2.2 for a way to circumvent this limitation).

A way to state the initial goal for the 3-queens problem was proposed in example 2.2.1. Following this proposal, the initial goal for the 4-queens problem is

\[
\text{place}(1) \land \text{place}(2) \land \text{place}(3) \land \text{place}(4)
\]

where \text{place} is user-defined by the definition\(^2\)

\[
\text{place}(\text{Col}) \leftrightarrow \text{queen}(\text{Col, 1}) \lor \text{queen}(\text{Col, 2}) \lor \ldots
\]

\(^2\)Of course, a recursive definition should be used to specify a general formulation of the \( n \)-queens problem.
So the initial goal can be unfolded to a conjunction of four disjunctions:

\[
\begin{align*}
& ( \text{queen}(1,1) \lor \ldots \lor \text{queen}(1,4) ) \land (D1) \\
& ( \text{queen}(2,1) \lor \ldots \lor \text{queen}(2,4) ) \land (D2) \\
& ( \text{queen}(3,1) \lor \ldots \lor \text{queen}(3,4) ) \land (D3) \\
& ( \text{queen}(4,1) \lor \ldots \lor \text{queen}(4,4) ) (D4)
\end{align*}
\]

The reader may find it helpful to draw a chessboard when following the derivation of goals by the proof procedure. Integrity constraints and any resolvents obtained from integrity constraints by propagating with them will be omitted from goals for simplicity.

First, the mark rule can be used to propagate, by local-to-local CPD, attacked atoms into all 16 disjuncts occurring in the goal. This corresponds to setting up 16 hypothetical chessboards with one queen each and crossing out all places to which the queen can move.\(^3\)

Splitting the first disjunct off the first disjunction makes queen(1,1) a global constraint in the simple goal

\[
\text{queen}(1,1) \land (D2') \land (D3') \land (D4')
\]

where (Di') is (Di) with added attacked atoms.

Global-to-local propagation can now be applied using the dead rule with the global constraint queen(1,1) and the local attacked constraints from each of the 12 disjuncts of the disjunctions (D2), (D3) and (D4). This operation may be seen as merging two chessboards and rejecting those merged boards (by generating false) on which queens are placed at coordinates that have already been crossed out.

After eliminating the disjuncts which correspond to rejected boards by logical equivalence transformations, the simple goal with global constraint queen(1,1) is reduced to

\[
\text{queen}(1,1) \land [ \\
\]

\(^3\)Note that these "chessboards" can be set up in parallel. In general, local-to-local CPD lends itself to parallel computation.
where the dots denote the attacked atoms generated in the initial local-to-local propagation phase.

Local-to-global CPD can now also be illustrated. In both disjuncts of the third disjunction, an atom attacked(2,3) has been generated by local-to-local CPD (since both queen(3,2) and queen(3,4) can move there), so the common local constraint attacked(2,3) can be promoted to a global constraint by local-to-global CPD (even in its simple form).

Global-to-local CPD with the dead rule using the new global constraint attacked(2,3) and the local constraint queen(2,3) from the first disjunct of the second disjunction adds false to this disjunct. Thus the disjunction representing the choices for the second column has been decided without splitting and collapses to a single disjunct, queen(2,4), which can be made a global constraint and used (by again applying global-to-local CPD with the dead rule) to similarly eliminate the queen(3,4) and queen(4,2) disjuncts, yielding the conjunction of now all-global constraints

queen(1,1) \land \text{queen}(2,4) \land \text{queen}(3,2) \land \text{queen}(4,3) \land \ldots

But queen(3,2) and queen(4,3) attack each other which is detected by global-to-global CPD (i.e. ordinary propagation) using the dead integrity constraint. Note that the required attacked atoms attacked(3,2) and attacked(4,3) were made global constraints (and are part of the "\ldots") when promoting the local constraints from the last remaining disjunct of the fourth, respectively third, disjunction to global constraints.

Thus it can be concluded that there is no solution with queen(1,1). Note that apart from the one initial splitting operation, no further splitting was required to come to this conclusion. In fact, only the first disjunction (D1) has to be split to identify the two solutions of the 4-queens problem if all four types of CPD are fully applied. The two solutions correspond to the two
computed answers

\[
\begin{align*}
&\text{queen}(1,2) \land \text{queen}(2,4) \land \text{queen}(3,1) \land \text{queen}(4,3) \\
&\text{queen}(1,3) \land \text{queen}(2,1) \land \text{queen}(3,4) \land \text{queen}(4,2)
\end{align*}
\]

What would have been different if the one integrity constraint

\[
\text{queen}(A,B) \land \text{queen}(C,D) \land \text{move}((A,B),(C,D)) \rightarrow \text{false}
\]

had been used instead of the two rules, mark and dead?

Firstly, it would have been advisable to assume that the definition of move is an enumeration of ground facts, i.e.

\[
\begin{align*}
&\text{move}((1,1),(1,2)) \leftrightarrow \text{true} \\
&\text{move}((1,1),(1,3)) \leftrightarrow \text{true} \\
&\ldots
\end{align*}
\]

so that any non-ground move atom is suspended (but see example 5.2.2 for a way to avoid this).

Secondly, many resolvents of the form

\[
\begin{align*}
&\text{queen}(C,D) \land \text{move}((a,b),(C,D)) \rightarrow \text{false} \quad \text{or} \\
&\text{queen}(A,B) \land \text{move}((A,B),(c,d)) \rightarrow \text{false}
\end{align*}
\]

with \(a, b, c, d\) denoting ground terms, would have been generated within disjuncts. Some of these resolvents correspond to the local attacked constraints in the two-rule solution. However, it is not possible to apply local-to-global CPD to these resolvents (not in its simple form anyway). One could avoid the generation of the resolvents by imposing a hyper-resolution restriction on propagation (see section 5.2.2 later in this chapter). Using this restriction and only global-to-local CPD, the one-rule solution is in fact quite efficient.

Some computational results for the two different approaches to the \(n\)-queens problem are given in section 6.2.2. For example, for \(n = 7\) the application of local-to-global CPD in the two-rule approach reduces the number of necessary splitting operations to find all solutions by about a third.
5.1.4 Other techniques for handling disjunctive constraints

This section compares CPD with previously proposed methods for handling disjunctive constraints. Most of these methods can be described in terms of CPD at least informally. A formal definition of the other methods in terms of certain combinations of CPD methods is not attempted in this thesis, but could prove interesting (at least theoretically) as CPD might provide a unified approach to handling disjunctive constraints.

The following subsections are organized in a way similar to [JoSo93] where the different techniques are compared to Constructive Disjunction (which itself is discussed in subsection 5.1.4.5).

5.1.4.1 PROLOG choice point

The disjunctive scheduling constraint

\[(t_1 + p_1 \leq t_2) \lor (t_2 + p_2 \leq t_1)\]

from section 5.1.1 could also be represented as two clauses of a normal logic program

\[
schedule( T_i, P_i, T_j, P_j ) \leftarrow T_i + P_i \leq T_j
\]

\[
schedule( T_i, P_i, T_j, P_j ) \leftarrow T_j + P_j \leq T_i
\]

In the CALOG framework, the two clauses can be merged into one if-and-only-if definition with a disjunction in its body. PROLOG's non-deterministic search over the choice point corresponds to not applying any of the CPD methods (not even global-to-global CPD, i.e. ordinary propagation, since PROLOG's only operation is unfolding), but instead to splitting the disjunction as soon as the definition has been unfolded and working on both disjuncts separately (using depth-first search in PROLOG's case).
5.1.4.2 Forward-checking and looking-ahead in CHIP

In the language CHIP [VH89] the same representation of the disjunctive constraint as in PROLOG can be used, but in addition CHIP allows the user to specify either a forward-checking declaration

\texttt{forward schedule(d,g,d,g)}.

or a look-ahead declaration

\texttt{lookahead schedule(d,g,d,g)}.

In both cases, the "d" signals that the first and third arguments of \texttt{schedule} are domain variables (for the task starting times), while the "g" signals that the second and fourth arguments of \texttt{schedule} are always ground (the processing times).

In general, a forward-checking declaration is used only if at most one of the arguments of a given constraint is a domain variable. This means that the given declaration for \texttt{schedule} can only be used after one of the starting times (the first or third parameter) has been determined. Only in this case will the constraint solver be activated to identify the remaining possible values for the other starting time. This is a rather strong requirement in the scheduling application because starting times may never be assigned a fixed value, but only confined to smaller and smaller intervals. If no values are left to satisfy the inequality constraint in one of the clauses, then the clause can be eliminated and the choice point is decided without non-deterministic search.

The \texttt{lookahead} declaration, on the other hand, tells the constraint solver to use the inequalities in the two clauses to propagate (internally) every update made to the domain of one starting time to that of the other. This is very similar to the combined use of global-to-local and local-to-local propagation (with explicit integrity constraints) in the CALOG framework.

The CALOG framework provides no direct way of simulating forward-checking, which is sometimes better than looking-ahead because the latter is computationally more expensive and pays off only if there are "strong inter-
actions between the constraints" [JoSo93]. Distinguishing between global-to-
local and local-to-local propagation may provide an elegant way of addressing
the efficiency problem because the distinction allows one to deactivate local-
to-local CPD while leaving global-to-local CPD active. A similar compromise
has been proposed in [MeMü93], which defines a technique called weak looking-
ahead as an intermediate solution which combines elements of both forward
checking and looking ahead.

5.1.4.3 The Andorra Model and AKL

Warren's Andorra Model (from "and-or") was originally proposed as a model
for parallel computation in logic programming. It has been applied to con-
straint programming in the Andorra Kernel Language (AKL) by Janson and
Haridi [JaHa91]. In the Andorra Model determinate subgoals are selected
before non-determinate subgoals. A disjunctive constraint is determinate if all
but one of its disjuncts has been shown to be unsatisfiable. A non-determinate
disjunctive constraint is only added to the constraint store if no determinate
disjunctive constraints are left.

If adding a non-determinate disjunctive constraint to the constraint store
results in an expansion of the search tree (i.e. splitting) and if the satisfiability
check used to determine whether a disjunctive constraint is determinate can be
implemented by global-to-local and local-to-local propagation, then Andorra
behaviour can be simulated in the CALOG framework: after exhaustively
applying global-to-local and local-to-local CPD, all remaining disjunctions are
non-determinate in the sense that the integrity constraints available to the
CPD methods cannot decide them and splitting has to be performed.

AKL also has a feature called encapsulated search which is discussed below
in section 5.1.4.6.

[Fr92] mentions AKL as a possible "host language" for CHRs (see section 4.3.5): CHRs
could be implemented in AKL and made available as a language extension.
5.1.4.4 The cardinality combinator of \texttt{cc(FD)}

\texttt{cc(FD)} [VHSaDe93] provides the \textit{cardinality combinator} for specifying disjunctive constraints:

\[
\#(\, l, u, [\ c_1, \ldots, c_n \ ] \ )
\]

holds if at least \( l \) and at most \( u \) of the \( n \) constraints \( c_1, \ldots, c_n \) are true in the current constraint store. The cardinality combinator can be used to describe various logical connections between constraints, e.g. setting the lower bound \( l \) to 1 and the upper bound \( u \) to \( n \) yields a disjunction

\[ c_1 \lor \ldots \lor c_n \]

Setting both \( l \) and \( u \) to \( n \) yields a conjunction

\[ c_1 \land \ldots \land c_n, \]

Setting both \( l \) and \( u \) to 0 yields a negation

\[ \neg c_1 \land \ldots \land \neg c_n. \]

So the schedule predicate can now be defined as

\[
\text{schedule}(\ T_1, P_1, T_2, P_2 \ ) \leftarrow \\
\#(\ 1, 2, [\ T_1 + P_1 \leq T_2, T_2 + P_2 \leq \ T_1 \ ] \ ).
\]

The cardinality combinator enables \texttt{cc(FD)} to decide the disjunctive constraint as soon as one of the two alternative schedules becomes unsatisfiable. It is not clear from the description in [VHSaDe93] how this is checked by the constraint solver; a full satisfiability check would effectively give looking ahead behaviour. [VHSaDe93] says only that "constraint entailment is used in a local manner to determine if the cardinality expression has a solution" and that "entailment is approximated through domain and interval entailment". In practice, this probably means for the scheduling example that consistency of each disjunct with the lower and upper bounds (or "domains") of the starting times is checked, i.e. it is checked whether the minimum possible value of \( t_1 \), plus the processing time \( p_1 \) is smaller than or equal to the maximum possible value of \( t_2 \).

This is effectively the same as global-to-local (and, if necessary in other applications, local-to-local) propagation with the integrity constraints used in
the examples above. As part of a no-box approach to constraint solving (see section 4.3.2), CPD may be more flexible since the user can specify the integrity constraints which are most suitable for a particular application.

5.1.4.5 Constructive Disjunction

*Constructive Disjunction* is also implemented in cc(FD) and informally described in [VHSaDe93], but a more detailed description together with a formalization can be found in [JoSo93]. Constructive Disjunction is based on the same idea as local-to-global CPD, i.e. to extract information common to all disjuncts out of a disjunction and thus to promote local constraints to global constraints.

[JoSo93] propose two different ways of implementing Constructive Disjunction (and thus local-to-global propagation). The first is called *local look-ahead reduction*. It computes for each variable and each disjunct of a disjunctive constraint those domain values which yield an inconsistency. The intersection of all these values is then the set of domain values which the variable cannot take regardless of which disjunct is chosen. For example, if the disjunction is \( t \leq 5 V \ 10 \leq t \), then the set of inconsistent domain values for \( t \) in the first disjunct is \((5, \infty)\) and in the second disjunct \((-\infty, 10)\), so the intersection of the two inconsistent domains is the open interval \((5, 10)\) (cf. the example given in section 5.1.1.4).

[JoSo93] remark that local look-ahead reduction is an efficient technique when dealing with inequality constraints, but may be computationally too expensive for other types of constraints. They therefore also introduce *global look-ahead reduction*. This technique first computes the (local) domains of all variables in each disjunct, taking into account all variables simultaneously. (This could be done by global-to-local and local-to-local propagation.) Then the union of all domains of each variable is taken over all disjuncts to obtain the possible (global) domain values for every variable in the disjunction. In the example of the disjunction \( 5 \leq t \lor t \leq 10 \), the domain of \( t \) in the first disjunct
is \([5, \infty)\), in the second disjunct \((-\infty, 10]\). The union of these two intervals is the set of all integers (or reals, depending on the underlying domain) excluding the interval \((5, 10)\), which is the same result as with local look-ahead reduction (as it should be).

The two different strategies for implementing Constructive Disjunction described above can be understood as different ways of implementing local-to-global CPD. By allowing the user to explicitly specify integrity constraints for this purpose, the CALOG framework already provides some flexibility, but more control measures may have to be added in an implementation to achieve good results for different classes of problems.

A technique closely related to Constructive Disjunction is Generalized Propagation \([LPWa92]\). While the formalization is different, the actual differences turn out to be rather small and technical. For the purposes of this comparison, Generalized Propagation may also be understood as a way of implementing local-to-global propagation.

5.1.4.6 Encapsulated search in Oz

Oz \([Sm95, HeSmWü95]\) is a concurrent constraint programming language (cf. section 4.4.2). Of interest in the current context is its \texttt{solve} combinator for \textit{encapsulated search}. It may be regarded as an extension of a similar feature in AKL (section 5.1.4.3).

In Oz, \textit{blackboards} are used to store constraints, and the \texttt{solve} combinator creates a new blackboard for each disjunct of a disjunctive constraint. The global constraints (from the global backboard) are copied to the local blackboards so that constraint solving can take place using all relevant information. Using global-to-local and local-to-local propagation has the same effect (but does not require constraints to be copied). The Oz approach may come closest, among the related approaches to handling disjunctive constraints discussed in this section, to the way global-to-local and local-to-local CPD have been defined. However, the formalization of Oz is very different from the CALOG
framework and also from most other (C)CLP approaches.

5.2 Other improvements of propagation

This section points out some problems and inefficiencies of the propagation operation and discusses a number of possible amendments. There remains scope for further research, and a practical evaluation of the proposed techniques for improving efficiency appears desirable. This would be facilitated by a complete and optimized implementation of the CALOG framework and its proof procedure (which is not yet available, but see chapter 6).

5.2.1 $P_1$-resolution

A major source of inefficiency in the CALOG proof procedure is any application of propagation where the Susp part of definition 3.1.5 is not empty, i.e. where two genuine implications are resolved with each other (rather than one atom and one implication). Such propagation may not only lead to the generation of resolvents (also called residues) which are never used, but also to non-termination of the proof procedure. This was illustrated in example 3.4.1 where the transitivity rule for "\(>\)" was part of the integrity constraints and where propagation between transitivity and an implication in the goal led to the generation of an infinite sequence of implications:

\[
A > Y \land Y > 2 \rightarrow \text{false}
\]

\[
A > Y' \land Y' > Y \land Y > 2 \rightarrow \text{false}
\]

\[
A > Y'' \land Y'' > Y' \land Y' > Y \land Y > 2 \rightarrow \text{false}
\]

\[
\ldots
\]

It is well-known in the theorem-proving literature (see [ChLe73, EiOh93]) that resolution stays refutation complete if one of the two resolvents has to be a positive clause (i.e. an atom or a disjunction of atoms). This resolution strategy is called $P_1$-resolution. It is easy to adapt definition 3.1.5 so that propagation is restricted to $P_1$-resolution: it suffices to require Susp to be either absent or
identical to true.

Since $P_1$-resolution is refutation complete, termination of propagation (without generating \texttt{false}) ensures consistency between a computed answer and the integrity constraints. This ensures that the relevant condition of the weak answer definition, condition (\(A3_w\)), is satisfied and hence the weak affirmation soundness theorem 3.3.1 continues to hold if propagation is restricted to $P_1$-resolution.

However, $P_1$-resolution may lead to the generation of computed answers which are not good answers and which would not have been computed by unrestricted propagation. This is illustrated by example 3.1.1 on which example 3.4.1 is based. Example 3.1.1 showed that it may be necessary to resolve two implications to detect an inconsistency with $T' \cup IC$: in the two implications
\[
\begin{align*}
1 > 0 & \rightarrow \text{false} \\
1 > Z & \rightarrow A > Z
\end{align*}
\]
all atoms are suspended, so unfolding cannot be applied, and under the $P_1$-resolution restriction the two implications cannot be used for propagation either. Unrestricted propagation, however, yields
\[
1 > 0 \rightarrow \text{false}
\]
which contains the reducible atom $1 > 0$ and thus makes the inconsistency detectable.

Example 4.2.2 used example 3.1.1 to point out why Fung's Iff Proof Procedure (IPP) does not need to allow propagation between two implications to achieve consistency with both the theory and the integrity constraints: because it never suspends any atoms. For analogous reasons, $P_1$-resolution does not lead to the generation of (non-good) weak answers in the ALP and SQO cases or, in general, whenever all definitions of user-defined predicates are written in homogenized form and when equality is the only built-in predicate. Therefore, propagation could be restricted to $P_1$-resolution in the stronger affirmation soundness result theorem 3.3.2.
5.2.2 Hyper-resolution

Resolution still remains refutation complete if all condition atoms of an implication have to be resolved away at the same time — the corresponding resolution strategy is called hyper-resolution (again, see [ChLe73, EiOh93] or other books on theorem-proving).

While restricting the definition of propagation to \( P_1 \)-resolution was straightforward, a similar definition for hyper-resolution is complicated by the possible occurrence of user-defined or built-in condition atoms which cannot be resolved away, but have to be unfolded after propagation has been applied.

Definition 5.2.1 (Propagation by hyper-resolution)

Assume that \( G \) is a conjunction (e.g. a simple goal) containing the implication

\[ \text{Cond} \rightarrow \text{Conc} \]

and that

\[ p_1(X_1), \ldots, p_n(X_n), \ n \geq 1 \]

are all the atoms of external predicates appearing in \( \text{Cond} \), with \( X_1, \ldots, X_n \) being vectors of terms. Let \( \text{Cond}' \) be the conjunction of any other atoms occurring in \( \text{Cond} \). If

\[ p_1(Y_1), \ldots, p_n(Y_n) \]

occur as (not necessarily mutually distinct) conjuncts in \( G \),\(^5\) then propagation by hyper-resolution adds the hyper-resolvent

\[ X_1 = Y_1 \land \ldots \land X_n = Y_n \land \text{Cond}' \rightarrow \text{Conc} \]

as a conjunct to \( G \).

In practice, the definition could be refined by immediately applying equality rewriting and unfolding to the hyper-resolvent. One might prefer not to add the hyper-resolvent to the goal if these operations do not manage to reduce its condition to true (thus simulating "real" hyper-resolution which completely

\(^5\)Strictly speaking, implications with condition true have to be considered as well. Remember that transforming such an implication into an atom (or a disjunction of atoms, depending on what the conclusion of the implication is) may be an unsafe operation, cf. definition 3.1.8, which is why such implications may remain in the goal.
eliminates the conditions) so that Conc can be added to the goal directly. This strategy has been applied in the implementation (chapter 6). It can be understood as a way of separating the condition of an integrity constraint into one part (of external atoms) which is matched against the goal and a guard part (of user-defined and built-in atoms) which has to be evaluable to true (by unfolding and equality rewriting), cf. section 4.3.4 and see section 5.2.5 below for an attempt to formalize this idea.

The following example illustrates how $P_1$- and hyper-resolution constrain the possible choices for propagation.

**Example 5.2.1**

$G_0$: $a$

$IC : a \rightarrow b$

$a \land b \rightarrow c$

If no restriction is imposed on propagation, then the two integrity constraints could be resolved with each other (on $b$), yielding

$a \land a \rightarrow c$

which requires two identical steps of further propagation to derive $c$ (unless one of the two $a$ atoms in the condition is removed by factoring or subsumption, cf. section 5.3).

Alternatively, if the $P_1$-resolution restriction was imposed, the proof procedure could have started by applying propagation between the goal atom $a$ and the second integrity constraint. This yields the residue $b \rightarrow c$ which cannot be immediately used for further propagation.

If instead the hyper-resolution restriction had been enforced, then the only possible step of propagation would have been between the goal atom $a$ and the first integrity constraint, yielding the new goal atom $b$. Now both condition atoms of the second integrity constraint are available and hyper-resolution can be applied between $a$, $b$, and $a \land b \rightarrow c$, generating $c$ in one step. The intermediate residue $b \rightarrow c$ is thus avoided.
Hyper-resolution may avoid the generation of many residue implications which would be generated by ordinary propagation or by \( P_1 \)-resolution. This can have both advantages and disadvantages. If an implication has many condition atoms, then finding that all but one of them can be satisfied may require a substantial computational effort which may have to be redone (when the last condition atom becomes available) unless a residue is generated to save the information. On the other hand, the more residues there are in a goal the more implications have to be stored (which may require not only memory resources but also some administrative effort) and checked for the applicability of propagation. Moreover, if an integrity constraint has defined atoms in its condition which are not unfolded before generating residues by \( P_1 \)-resolution, then they may have to be unfolded several times. These problems of \( P_1 \)- and hyper-resolution will be addressed in the section on locking resolution below.

Hyper-resolution probably corresponds closest to the procedural interpretation of integrity constraints as forward propagation rules ("if the conditions hold, then derive the conclusion", cf. the introduction) and may be what the user really had in mind when writing down such rules as integrity constraints. Examples will be provided in the applications chapter 6.

### 5.2.3 Locking resolution

Another possible restriction of propagation, originally proposed for the IfP Proof Procedure in [Fu96], is the identification of a *pivotal abducible atom* in the implications occurring in a goal. This corresponds to the use of a *locking resolution strategy* (see [ChLe73, EiOh93]). In general, locking resolution allows an ordering of the literals in a clause which determines the order in which the literals may be resolved away (and the residues inherit the ordering of the parent clauses). Identifying a pivotal abducible atom in every implication corresponds to a locking strategy in which the abducible atoms are given highest (but mutually distinct) priorities whereas all non-abducible atoms are given (the same) lower priority.
The following definition adapts the concept to the CALOG proof procedure.

**Definition 5.2.2** Let \( \text{Cond} \rightarrow \text{Conc} \) be an implication occurring in a goal such that \( \text{Cond} \) contains at least one atom of an external predicate. Let \( A \) be the first such atom (from left to right). Then \( A \) is called the **pivotal external atom** of the implication.

Taking the leftmost external atom as the pivotal atom is an arbitrary choice; any external atom in the condition could be chosen.

The proof procedure can now be restricted in two ways. If an implication \( I \) occurring in a goal has a pivotal atom \( A \), then:

- unfolding of atoms in the condition of \( I \) is not permitted
- propagation with \( I \) is only permitted if resolution is performed on \( A \)

Both of these restrictions are justified as instances of locking resolution which, like \( P_1 \)- and hyper-resolution, is a refutation complete resolution strategy. While it has the disadvantage of being incompatible with other restrictions on resolution (in the sense that refutation completeness may be lost even if the other restriction is also refutation complete; this is the case for a combination of hyper-resolution and locking, for example), locking by means of pivotal external atoms has the advantage of being fully compatible with suspension: it does not lead to the computation of weaker answers which would have been rejected by unrestricted propagation. But note that using pivotal external atoms in example 3.4.1 has no effect because there are no external atoms in the implications.

Moreover, locking resolution can also be used to "implement" \( P_1 \)- and hyper-resolution in an efficient way. By identifying an order in which the literals in implications have to be resolved away (by several steps of \( P_1 \)-resolution), the advantages of \( P_1 \)- and hyper-resolution can be combined while the problems cited at the end of the previous section can be avoided.\(^6\) It is no longer

---

\(^6\)This strategy was first proposed in [KoHa69], although it was not referred to as locking resolution.
possible to generate numerous different residues from one implication (possibly repeating the same operations several times) and it is neither necessary to abandon an incomplete attempt at applying hyper-resolution.

5.2.4 Propagation histories

Another addition to the proof procedure proposed in [Fu96] is the attachment of a propagation history to every atom occurring in the condition of an implication. Initially, the set of implications occurring in the goal is the set of integrity constraints, and the propagation history of the condition atoms in every integrity constraint is empty.

Propagation is then refined to check the propagation history of the resolved atom in the condition of an implication for occurrence of the atom with which it is to be resolved. If the other atom does occur in the propagation history, then propagation is not permitted.

Note that this behaviour has been tacitly assumed for the CALOG proof procedure as far as repeating propagation between exactly the same expressions in a goal is concerned, i.e. if

\[ a \land [a \rightarrow b] \]

have been used to propagate b into the goal, then the two will not be used again for propagation (even if b disappears from the goal due to unfolding). To implement this behaviour is trivial.

However, the introduction of another atom a into the goal (by propagation with other implications) would trigger one step of propagation between the new occurrence of a and the (old) implication a → b. If this is to be avoided, then propagation histories are needed. Alternatively, however, factoring (in the IPP) or subsumption (see secton 5.3) could be used to eliminate the new duplicate atom immediately (the case that this is not possible because the old atom has disappeared due to unfolding could be excluded by disallowing propagation between reducible atoms — another sensible restriction on propagation). It seems that [Fu96] introduced propagation histories mainly as a
theoretical device to exclude the trivial case of a repetition of propagation between exactly the same two (old) expressions.

5.2.5 Guarded integrity constraints

Guarded rules have been used in [Sm91] (see section 4.3.4) and [Fr92] (see section 4.3.5) for frameworks similar to the CLP instance of the CALOG framework. The idea of treating some conditions of integrity constraints as guards has already been mentioned in the section on hyper-resolution above. There it was suggested that atoms of user-defined predicates and of equality — or, in generalization of the idea and in correspondence to the usual restriction in CLP that guards be constraints: of any built-in predicate — should be evaluable to true and that no resolvent should be added to the goal if they are not.

One could add some flexibility to this concept by allowing the user to explicitly identify some condition atoms of some integrity constraints as guards.

Definition 5.2.3 A guarded implication is an implication in which zero or more, but not all condition atoms have been identified as guards.

Syntax: \( C_1 \land \ldots \land C_n | G_1 \land \ldots \land G_m \rightarrow \text{Conc}, n \geq 1, m \geq 0 \)

\( G_1, \ldots, G_m \) are the guards which have to be atoms of user-defined or built-in predicates.

The definitions of propagation and unfolding have to be adapted to take guards into account, and there are various options how to do this. Arbitrary unfolding of reducible guards seems to defy the purpose for which guards are mainly introduced (i.e. increasing efficiency). Similarly to the treatment of guards in CLP, it seems appropriate to have a separate evaluation operation for guards which unfolds reducible guards with only two possible outcomes: removal of a guard atom from the implication if the guard atom could be unfolded to true or removal of the implication from the goal if a guard atom
could be unfolded to false. If neither outcome is possible, then a guard cannot be evaluated.

Since propagation with a guarded rule should only be permitted if all guards can be evaluated to true, using a variant of hyper-resolution for guarded integrity constraints seems sensible. Based on the definition of propagation by hyper-resolution, propagation with guarded implications could thus be defined as follows.

**Definition 5.2.4 (Guarded propagation)**

Assume that a simple goal $G$ contains the guarded implication

$$\text{Cond} \mid \text{Guard} \rightarrow \text{Conc}$$

and that

$$p_1(X_1), \ldots, p_n(X_n), n \geq 1$$

are some atoms appearing in $\text{Cond}$, with $X_1, \ldots, X_n$ being vectors of terms. Let $\text{Cond}'$ be the conjunction of any other atoms occurring in $\text{Cond}$. If

$$p_1(Y_1), \ldots, p_n(Y_n)$$

occur as (not necessarily mutually distinct) atoms in $G$, then *guarded propagation* adds the resolvent

$$\text{Eq} \land \text{Cond}' \rightarrow \text{Conc}$$

as a conjunct to $G$, provided that

$$X_1 = Y_1 \land \ldots \land X_n = Y_n \land \text{Guard}$$

can be reduced, in a finite number of steps, to a conjunction $\text{Eq}$ of zero or more equalities by exhaustive application of equality rewriting and evaluation.

Alternatively, the definition could be based on the locking resolution strategy outlined in section 5.2.3 above (rather than on hyper-resolution).

To avoid unexpected results (like non-termination of guard evaluation), restrictions could be imposed on the user-defined predicates permitted in guards. For example, their definitions could be required to be enumerations of ground facts. On the other hand the following example shows a useful side effect which the use of guards without this restriction may have.
Example 5.2.2 (Guards for n-queens)

The integrity constraint

\[ \text{queen}(A, B) \land \text{queen}(C, D) \land \text{move}((A, B), (C, D)) \rightarrow \text{false} \]

can be transformed into a guarded rule

\[ \text{queen}(A, B) \land \text{queen}(C, D) \mid \text{move}((A, B), (C, D)) \rightarrow \text{false} \]

In this formulation move could be defined without enumerating all ground facts as the guard may be reducible but is not evaluable until the queen atoms provide the necessary variable bindings.

The use of guards in integrity constraints is different from the use of guarded rules in concurrent programming languages (cf. section 4.4) where programs can be made determinate if every predicate is defined by clauses with mutually exclusive guards. A similar extension of the CALOG framework, i.e. adding guards to if-and-only-if definitions, is also conceivable, but is left for future research together with the general problem of incorporating concurrency.

5.3 Subsumption

Subsumption is an operation which eliminates logically redundant implications and atoms, thus avoiding unnecessary computations involving them. Subsumption could be added as a further operation to the proof procedure. Some of the many uses of subsumption are outlined in this section.\(^7\)

**Definition 5.3.1** The implication \( A_1 \land \ldots \land A_m \rightarrow B_1 \lor \ldots \lor B_n \) subsumes the implication \( C_1 \land \ldots \land C_r \rightarrow D_1 \lor \ldots \lor D_s \) if there is a substitution \( \sigma \) such that

\[
\{ A_1 \sigma, \ldots, A_m \sigma \} \subseteq \{ C_1, \ldots, C_r \} \quad \text{and} \\
\{ B_1 \sigma, \ldots, B_n \sigma \} \subseteq \{ D_1, \ldots, D_s \}.
\]

The definition is meant to also apply to atoms as degenerate implications

\(^7\)The definition of subsumption used here is not the standard definition, but is equivalent to it. Partial subsumption, which is a different operation but based on the idea of subsumption, was already defined in the SQO review (section 4.1.2.1).
\[(m = 0, n = 1 \text{ and } r = 0, s = 1, \text{ respectively}).\]

The subsumption operation removes subsumed implications (or atoms) whenever it detects them. [Fu96] proposes to use only unit subsumption, i.e. to test only whether an atom (the case \(m = 0, n = 1\) in the definition) subsumes an implication. He argues that a full implementation of subsumption would be computationally too expensive. This might be worth testing in practice.

A test for subsumption after a step of propagation may also be useful because a resolvent sometimes subsumes one of its parents. This is always the case if a ground atom is resolved with a ground implication, e.g.

\[
a \land [a \land b \rightarrow c]
\]
yields, by propagation

\[
a \land [a \land b \rightarrow c] \land [b \rightarrow c]
\]
The residue \(b \rightarrow c\) subsumes the parent clause \(a \land b \rightarrow c\), which could thus be removed.

[Fu96] contemplates extending his use of propagation (which is otherwise restricted to the case in which an atom is resolved with an implication, cf. section 4.2.2.3) to resolution between two implications provided that the resolvent subsumes at least one of the parent clauses. This may be a good way of allowing some resolution between implications while avoiding the non-termination problem (caused by unrestricted resolution between implications) which was illustrated in example 3.4.1.

Removing subsumed implications not only increases efficiency, it also simplifies computed answers. In the example above, the computed answer without applying subsumption contains the atom \(a\) and two denials

\[
b \rightarrow \text{false}
\]
\[
a \land b \rightarrow \text{false}
\]
of which the second is subsumed by the first and thus redundant. The use of subsumption could even be restricted to the simplification of computed answers (if a general use is deemed too costly).
Recall that subsumption can also be used to simulate (together with addition of tautologies and other equivalence transformations) factoring (section 4.2.2.6) and case analysis (section 4.2.2.7). This further underlines the fact that subsumption is a versatile operation, and its addition to the proof procedure might be considered even before these other operations (which it "subsumes", at least partly).

Yet another use of subsumption — for the deletion operation to be presented next — is analysed in section 5.4.2.

5.4 Deletion

5.4.1 The deletion operation

Subsumption provides only limited possibilities of removing useless information from the goal. Frameworks in which more general methods are used to simplify goals have already been mentioned in chapter 4: in the transformation phase of SQO (see section 4.1.2.2), atoms are sometimes deleted from the query because it is known that they will simply succeed (without providing any further variable bindings); in Smolka's CLP framework (see section 4.3.4), guarded equivalences are used as rewrite rules; Frühwirth defines both propagation CHRs, which are like integrity constraints, and simplification CHRs, which, like Smolka's guarded equivalences, add new information and remove old information at the same time (see section 4.3.5).

In all these cases the old information which is removed does not have to be removed, but doing so increases efficiency as goals get simpler and fewer operations become applicable. To achieve similar effects for the CALOG proof procedure, a deletion operation can be added to it. In its most general form, deletion could be defined analogously to ordinary propagation (definition 3.1.5) as resolution between two implications (or an atom and an implication), with the difference being that the resolvent is not added to the goal, but instead removed from the goal.
However, in practice a restricted form of deletion, which corresponds to guarded propagation (definition 5.2.4) by hyper-resolution (section 5.2.2) seems more adequate:

**Definition 5.4.1 (Deletion)**

Assume that a simple goal $G$ contains the integrity constraint

$$p_1(X_1) \land \ldots \land p_n(X_n) \land Cond' \rightarrow A$$

where $Cond'$ is a conjunction of atoms of user-defined and built-in predicates and where $A$ is an atom. Let there be a substitution $\sigma$ such that $A\sigma$ appears as a conjunct in $G$. Assume further that

$$\begin{align*}
MY, & \quad \ldots, p(Y_n) \\
\end{align*}$$

occur as (not necessarily mutually distinct) atoms in $G$ and that

$$\begin{align*}
X_1\sigma = Y_1\sigma & \land \ldots \land X_n\sigma = Y_n\sigma \land Cond'\sigma \rightarrow A\sigma \\
\end{align*}$$

can be transformed, in a finite number of steps of equality rewriting and evaluation (see section 5.2.5), to

$$\begin{align*}
true & \rightarrow A\sigma \\
\end{align*}$$

Then $A\sigma$ can be removed from $G$.

The following example illustrates the definition.

**Example 5.4.1** Deletion is useful in the scheduling application (cf. section 5.1.1) to get rid of redundant bounds on starting times after propagating tighter bounds, e.g. if the old bound on $t$ was

$$t > 5$$

and the new bound is

$$t > 10,$$

then transitivity

$$X > Y \land Y > Z \rightarrow X > Z$$

can be used to delete the now redundant old bound $t > 5$. In this case the atom $A$ from the definition of deletion is $X > Z$ and the substitution $\sigma$ is $(X/t, Z/5)$. The first condition, $X > Y$, is matched with $t > 10$, yielding (after application of the substitution $\sigma$)
Equality rewrite rule (EQ3) gets rid of \( t=t \). (EQ6) replaces the two occurrences of \( Y \) by 10. The two resulting atoms, \( 10=10 \) and \( 10>5 \), can be evaluated to true, so deletion may be applied and \( t>5 \) is removed from the goal.

Like subsumption, deletion is used to eliminate logically redundant information. It is thus a sound operation in the sense of lemma 3.3.1 (goals derived from each other by one step of deletion are equivalent) and could be added to the proof procedure without affecting any soundness results.

Propagation and deletion have to be operationally restricted so that they do not undo each other's effects, possibly causing infinite loops. This can be done either by the user who could partition the set of integrity constraints into two disjoint sets of propagation rules and deletion rules (which are both simply integrity constraints designated for the respective purpose). However, some integrity constraints, such as transitivity, may at some times be usable for propagation and at other times for deletion.9

Moreover, to achieve the operational behaviour of guarded equivalences or simplification CHRs, propagation and deletion rules could be coupled: if a propagation rule has been applied, the associated deletion rules are tried immediately.

5.4.2 Deletion by subsumption

Deletion can sometimes be understood in terms of subsumption (cf. section 5.3). This observation, which is illustrated by the following example, could prove useful in identifying ways of implementing and automating deletion, at least in some cases.

Example 5.4.2 Consider again the transitivity integrity constraint for "\( >\)".

---

8This implication is not added to the goal. It is shown here only to illustrate the definition.

9Ideally, the proof procedure should be able to decide for itself when an integrity constraint is to be used for deletion and when for propagation. This could be another interesting topic for future work.
\[ \text{If } Y > Z \text{ holds, then the information } X > Z \text{ is useless if it is also known that } X > Y, \]
and the transitivity rule can be used to delete the useless information. Now note that \( X > Z \) is equivalent to \( X \) being either greater than \( Z \) but smaller or equal than \( Y \), or greater than \( Y \), i.e.
\[
X > Z \leftrightarrow X \in [Z, Y] \lor X > Y
\]
published that \( Y > Z \) holds. Since
\[
X > Y
\]
subsumes
\[
X \in [Z, Y] \lor X > Y
\]
subsumption can be used to delete this disjunction (and thus \( X > Z \), which is equivalent to the disjunction).

It is also instructive to consider the case where \( X \) is restricted to a finite domain of values, e.g. \( X \in [n, m] \). This information can be expressed by the disjunction
\[
X = n \lor \ldots \lor X = m
\]
Now suppose the constraint \( X > k \) becomes available (e.g. by propagation), where \( k \) is an integer with \( n \leq k < m \). The domain of \( X \) can now be represented by the new disjunction
\[
X = k + 1 \lor \ldots \lor X = m
\]
which subsumes the old one (since \( k > n \), at least one disjunct from the old disjunction is missing).

The example shows that, if the finite domain of a variable is represented by a disjunction, then reducing the domain corresponds to removing disjuncts from the disjunction which represents the domain. Thus the resulting disjunction subsumes the original one. The example also shows that intervals can be used to finitely represent infinite domains as disjunctions so that the same subsumption strategy can be used to simulate deletion in infinite domains.

Note that this interpretation of some deletion operations in terms of subsumption is not restricted to numerical domains. Any finite domain can be
represented by a disjunction. Sometimes a list representation may be more adequate, i.e.

\[ \text{member}(X, l) \]

could be used to indicate that the list \( l \) contains all possible domain values for \( X \). Constraining the domain then corresponds to removing elements from \( l \) and obtaining the new constraint

\[ \text{member}(X, l') \]

where \( l' \) is a sublist of \( l \). Then

\[ \text{member}(X, l') \land \text{sublist}(l, l') \rightarrow \text{member}(X, l) \]

So \( \text{member}(X, l) \) is redundant and can be deleted.

5.4.3 Production rule systems

As already mentioned in section 4.3.4, Smolka [Sm91] defines both admissible and non-admissible guarded rules. Whereas admissible guarded rules can be understood as integrity constraints which are properties of the definitions, non-admissible guarded rules correspond to any implications which are not properties of the definitions, but which are nevertheless used for propagation and deletion. Such operations cannot be proven sound in the sense that they preserve equivalence in the intended models of the theory (cf. corollary 3.3.2), but might still be useful in practice, especially to simulate the behaviour of production rule systems (see [Wi92] for a general introduction to this topic).

Raschid and Lobo [Ra94, RaLo94] propose to give a logical semantics to special classes of production rule systems. They distinguish between two types of production rules:

- **a-productions**, which are rules of the form

  \[ A \rightarrow \text{assert} \]

  where each \( L_i \) is a literal, and \( A \) is an atom

- **r-productions**, which are rules of the form
\[ L_1 \land \ldots \land L_n \rightarrow \text{retract } A \]

where again each \( L_i \) is a literal, and \( A \) is an atom

Both a- and r-productions have to be range-restricted in the sense that every variable occurring in a production rule must occur in a positive literal \( L_i \) (cf. definition 3.1.9).

From a purely operational point of view, a-productions work like propagation rules and r-production like deletion rules: propagation (by hyper-resolution, cf. section 5.2.2) can be understood as using an integrity constraint like an a-production whereas deletion can be understood as using an integrity constraint like an r-production.

Consequently, the CALOG proof procedure augmented with deletion rules could be used to execute production rule systems. However, for most production rule systems simply translating a- and r-productions into integrity constraints for propagation and deletion would not obey the CALOG framework's restriction that integrity constraints have to be properties of the theory \( T \) and thus computed answers would not be semantic answers according to definition 2.3.2.

Raschid and Lobo propose to transform a given production rule system (i.e. a set of a- and r-productions) into a normal logic program. For certain classes of production rule systems they prove that a generalization of the well-founded semantics (cf. section 2.5), yields the same models for the resulting normal logic program as are computed by a fixpoint operator applied to the original set of production rules. It might be worth investigating whether a translation of production rules into definitions and integrity constraints could yield a similar result (with the possible advantage that, thanks to the use of integrity constraints, the resulting representation is closer to the original production rule system than the normal logic program obtained after Raschid and Lobo's transformation).
Chapter 6

Implementation and Applications

The applications presented in this chapter are taken from different areas of Artificial Intelligence and thus illustrate the wide utilizability of the CALOG framework. The proposed solutions should give the reader an idea of how to program efficiently in a new logic programming style combining definitions and integrity constraints. The solutions are all simple (short and easy to write), natural (close to the problem specification) and flexible (easy to adapt to special requirements of particular problems), even for “real-world” problems like Operations Research applications (see section 6.3).

Programs written in this new style should prove more efficient than programs written in standard LP style, although standard LP may sometimes be able to achieve similar efficiency at the cost of sacrificing simplicity and naturalness of the solution (e.g. see section 6.5.2).

It appears unlikely that solutions purely programmed in the CALOG framework can beat black-box CLP (cf. section 4.3.2) when used for the types of problems for which black-box constraint solvers were designed. Therefore, integrating a black-box solver for standard tasks (such as the solving of linear equations or systems of inequalities) seems advisable for efficiency purposes.
6.1 Implementation: PROCALOG

PROCALOG stands for *PROgramming with Constraints and Abducibles in LOGic* (or *PROgramming in CALOG*) and is an attempt to implement the CALOG framework and its proof procedure as a logic programming language. At present two implementations exist.

The first prototype implementation (of a predecessor of the CALOG framework) was part of the author's M.Sc. thesis [We94] and has been extended and refined afterwards. The implementation is done in (and on top of) PROLOG and could be considered a meta-interpreter. Unfolding and equality rewriting are not implemented, but rather left to PROLOG's SLD(NF) resolution and unification. Definitions have to be written as standard PROLOG clauses and no if-and-only-if is used. Disjunctions may be represented by atoms (e.g. or(A, B) could be used to represent A ∨ B), so use of CPD is possible and has been implemented, albeit only for binary disjunctions. The user has control over which CPD methods are to be used. Propagation has been implemented as hyper-resolution in the restricted and optimized form of guarded propagation (see definition 5.2.4). Guards do not have to be explicitly identified, however, as the system simply regards every atom of a defined predicate as a guard (whether defined by PROLOG clauses or as a PROLOG built-in). Deletion is also implemented, with the user having to identify integrity constraints either as propagation or as deletion rules (cf. section 5.4).

Obtaining a more efficient implementation was the aim of another M.Sc. project [Mn96]. The implementation, this time in Borland PASCAL (an extension of standard PASCAL which provides C-like functionality and efficiency), was carried out with the author's assistance, but the interpreter was completed only partially during the available time. In principle, all operations of the CALOG proof procedure (again including the CPD extensions) have

\[\text{An n-ary disjunction can of course be equivalently represented by n-1 binary disjunctions, and this has been done. However, the local constraints of disjuncts 2, \ldots, n are not available for propagation as long as } n > 2.\]
been implemented, but some of them do not fully work yet. Propagation is implemented in the same way as in the PROLOG prototype. To assist the interpreter in executing programs more efficiently, the user can specify domain declarations for user-defined predicates. For example,

\texttt{move(1..4, 1..4, 1..4, 1..4)}

can be used to tell the interpreter that the parameters of the predicate \texttt{move} (as used in the n-queens problem) can only take values from 1 to 4 (as in the 4-queens problem). This enables the interpreter to store the definition of the \texttt{move} predicate in a more efficient way (essentially in array form) so that the guard-like \texttt{move} atom in the integrity constraint

\begin{verbatim}
queen(A,B) \land queen(C,D) \land move((A,B), (C,D)) \landfalse
\end{verbatim}

can be evaluated (in the sense evaluation has been defined for the guarded propagation definition) by a simple indexed table lookup. Most PROLOG implementations index only on the first argument of predicates.

One could try to extend this concept (and thus further increase efficiency) by also allowing domain declarations for external predicates, e.g.

\texttt{queen(1..4, 1..4)}

could be used in the 4-queens problem to restrict the range of the parameters of the \texttt{queen} predicate. If properly and efficiently implemented, the interpreter might be able to achieve an operational behaviour almost identical to special purpose programs written in Pascal or C to solve only the n-queens problem. There does not seem to be any obvious theoretical obstacle to make it as efficient.

As mentioned already in section 3.1.3, the implementations adopt a rather simple search and ordering strategy. First exhaustive propagation is performed (including CPD), then one atom is unfolded (if there are any reducible atoms in the goal), then propagation is tried again, and so on, until every atom is suspended. Then a disjunction is split and processing continues, first on one disjunct, then on the other (i.e. a depth-first search strategy). Logical equivalence transformations (except splitting) and equality rewriting are always
performed as soon as they become applicable.

The PASCAL procedure which implements this strategy nicely formalizes the described algorithm:

```
PROCEDURE Apply_Operations( G : Goal );
...
REPEAT
  Propagate( G )
UNTIL NOT Unfold_one_atom( G );

IF Split( G, G1, G2 )
  THEN BEGIN { G=G1VG2 }
    Apply_Operations( G1 );
    Apply_Operations( G2 )
  END
ELSE
  Display_Solution( G )
```

It is not difficult to change this simple strategy or to allow the user to specify an ordering of the operations which may be more suitable for a particular problem. For example, it is often advisable to do several steps of unfolding first and delay propagation in order to make enough information available to achieve more useful results by propagation. This could be implemented by an unfolding loop before the REPEAT-loop which terminates either after exhaustively unfolding all reducible atoms in the goal (a sensible thing to do in many problems where the complete problem definition has to be unfolded first) or when an atom specified by the user has been introduced by unfolding (to trigger the next phase).

More control over propagation could be added. For example, it may be better to use only certain subsets of the integrity constraints at the same time,
if the user knows that some of them do not generate any helpful information (but may be applicable for propagation) during certain stages of the computation. Explicitly activating and deactivating integrity constraints is possible in the PROLOG prototype.

Before splitting a disjunction it is often useful to apply a heuristic to choose a disjunction. Such a heuristic could be included in the split procedure. Several general purpose techniques (cf. section 6.2.1) could be implemented and the user could choose among them. The user could even supply an external program implementing a special heuristic which is called whenever a disjunction is split.

Certain classes of problems may require other search techniques like breadth-first or iterative deepening. Implementing these poses no principle problem, but requires more programming effort. Optimization problems are usually solved using branch-and-bound or A* search. Implementing branch-and-bound is fairly easy and discussed in section 6.3.1.2. A* can then be realized as a combination of branch-and-bound and a heuristic for choosing the next disjunction to be split.

Some computational results obtained by both the PROLOG and the PASCAL implementation will be reported for the computationally more demanding applications. While the PROLOG prototype has been optimized as far as this was possible for a meta-interpreter approach, the PASCAL implementation is still rather incomplete and somewhat unreliable (and also far from being optimized, although it is already considerably faster than the PROLOG prototype for those problems which it can solve at present). For these reasons absolute running times (measured on a 486 PC with 33 MHz) have only very limited meaning, but the impact which the use of the CPD extensions has on both relative execution times and relative search space sizes may be of interest.
6.2 Constraint Satisfaction Problems

In a Constraint Satisfaction Problem (CSP) the values of \( n \) variables \( X_1, \ldots, X_n \), \( n \geq 1 \) are restricted to \( n \) finite domains \( D_1, \ldots, D_n \), respectively.\(^2\) The possible assignments of domain values to variables may be restricted by constraints which rule out or enforce certain combinations of variable assignments.

A generic initial goal for a CSP in the CALOG framework is
\[
\text{domain}(X_1, D_1) \wedge \ldots \wedge \text{domain}(X_n, D_n)
\]
where domain is defined by \( n \) if-and-only-if definitions
\[
\text{domain}(X_i, D_i) \leftrightarrow (X_i = v_{i1} \lor \ldots \lor X_i = v_{im})
\]
if \( D_i = \{v_{i1}, \ldots, v_{im}\} \).

For many CSPs, the constraints between the variables \( X_1, \ldots, X_n \) are specified in the form of implications (often denials) which can directly be rewritten into integrity constraints.

Frequently, different ways of specifying the initial goal and the constraints may be possible and may lead to different operational behaviour (cf. the 4-queens example in section 5.1.3).

6.2.1 Heuristic choices

In CLP, constraint satisfaction problems are usually solved by assigning values to variables (this is often called labeling) and then reducing the domains of the other variables according to the constraints. The choice of which variable to assign a value to next is often supported by a heuristic.

The standard strategies to support this choice (see e.g. [RuNo95]) include the most-constrained variable heuristic which chooses a variable with the fewest possible assignments and the most-constraining variable heuristic, which chooses a variable which occurs in the largest number of constraints (thus hopefully constraining the possible domains of many other variables).

\(^2\)This is the traditional definition which can be found, e.g., in [VH89]. In [RuNo95] problems with continuous domains are also called CSPs.
Not only the choice of the variable but also that of the value to be assigned to the chosen variable may matter (especially in optimization problems or in general whenever only one solution is to be found). The least-constraining value heuristic chooses a value which rules out the smallest number of assignments to other variables.

In the CALOG framework, assigning a value \( v \) to a variable \( X; \) corresponds to splitting off the disjunct \( X = v \) from the disjunction representing the domain of \( X \). It has already been mentioned in section 6.1 that incorporating heuristics (such as the ones mentioned above) for choosing a variable and a value (i.e. a disjunction and a disjunct) into the implementation would not be difficult and that the user could even supply a special-purpose external procedure to be called whenever a disjunct has to be chosen for splitting.

The most-constrained variable heuristic is particularly easy to implement: simply choose the disjunction with the fewest number of disjuncts. Since global-to-local and local-to-local CPD may help eliminate disjuncts, the heuristic can benefit from the application of these techniques. Also, to identify the least-constraining value the effects of different choices have to be precomputed. If appropriate integrity constraints are made available, then global-to-local and local-to-local CPD can have exactly this effect and the results of such propagation can be used by the heuristic (e.g. in the 2-rule formulation of the n-queens problem presented in section 5.1.3, the disjunct with the smallest number of attacked atoms could be chosen). Since the heuristics have proven to be very efficient in many applications, the CPD methods may also be able to increase efficiency (in spite of the computational overhead).

6.2.2 n-queens problem

The n-queens problem can be regarded as a CSP in which \( n \) variables \( Q_1, \ldots, Q_n \) (with \( Q_i \) representing the row number of the queen in column \( i \)) have to be assigned a value in the domain \( \{1, \ldots, n\} \) (the \( n \) domains are identical). The assignments are restricted by the constraint that no two queens may attack
each other according to the rules of chess.

[RuNo95] remark that using the most-constrained variable heuristic for the
n-queens problem increased the feasible problem size from $n = 30$ to about
$n = 100$. Additionally using the least-constraining value heuristic further
increases the feasible problem size to about $n = 1000$ (presumably for finding
the first solution, although this is not said explicitly). The most-constrained
variable heuristic has been added to the PASCAL implementation as an option
the programmer can activate before starting the computation.

The n-queens problem was presented in chapter 2 when introducing the
CALOG framework, and the derivation of solutions by the CALOG proof
procedure including the CPD extensions was discussed in chapter 5. Therefore
only computational results will be reported here (see the table below). Unless
otherwise indicated, all results are for finding all solutions (including the proof
that there are no other solutions).

The first two columns in the table specify the number of queens $n$ and
the number of solutions. The next columns give the processing times and
numbers of splitting operations for, respectively, the one-rule solution (see ex-
ample 2.1.1) without CPD (except for ordinary global-to-global propagation),
the one-rule solution with global-to-local CPD, the two-rule solution (see sec-
tion 5.1.3) with full application of all CPD methods including local-to-global
CPD, and the same when executed by the PROLOG prototype. The numbers
in brackets in the global-to-local CPD column are the numbers of splitting
operations when using the most-constrained variable heuristic.

<table>
<thead>
<tr>
<th>$n$</th>
<th>#S</th>
<th>1-r., no CPD</th>
<th>1-r., g-t-l CPD</th>
<th>2-r., full CPD</th>
<th>2-r., Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>0.4</td>
<td>0.3 5 (5)</td>
<td>2.0</td>
<td>2.9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1.4</td>
<td>1.4 13 (13)</td>
<td>13.3</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>7.3</td>
<td>5.7 39 (39)</td>
<td>56.0</td>
<td>99.5</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>37.5</td>
<td>25.8 111 (107)</td>
<td>547.0</td>
<td>n/a</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>210.4</td>
<td>132.4 415 (383)</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

The data in the table confirm the expectation that CPD is a powerful technique
which drastically reduces the search tree. Clearly, this is achieved at the expense of a higher computation time spent at each node in the search tree. First experiments with the PROLOG prototype had been slightly discouraging because the overall computation time sometimes increased significantly. Now the PASCAL implementation, even in its early stages, is able to obtain a genuine and non-trivial speed-up when global-to-local CPD is used (compare the first column of numbers in the one-rule, no CPD and the one-rule global-to-local CPD columns). Moreover, both the relative reduction in the number of nodes and the relative speed-up increase with growing problem size: for $n=6$ the search tree is shrunk to 5.2\% of its original size, for $n=7$ to 3.6\%, for $n=8$ to a mere 3.0\%. The speed-up is 22\%, 31\% and 37\%, respectively.

The additional reduction in the number of splitting operations obtained by using full CPD rather than only global-to-local CPD is promising, although the values should be interpreted with some care. For $n=6$, the number is almost halved, but this may be due to the special nature of the 6-queens problem which has only four solutions. For $n=7$, about a third of splitting operations are saved which is more similar to the result for $n=5$ (it may also be that the reduction is generally more significant for even values of $n$ than for odd values). The PASCAL implementation was not able to provide results for $n=8$ with full CPD and already used up almost all of the available memory (10 MB) for $n=7$. This and the rather drastic increase in computation time for full CPD are due to the preliminary nature of the implementation (the attacked atoms propagated into the goal are handled very inefficiently).

The last two columns in the table give times and splitting numbers for the PROLOG prototype, applying full CPD in the 2-rule solution. Not only were the processing times worse than in the generally faster PASCAL implementation, but the PROLOG prototype was also not able to take full advantage of the CPD methods as only binary disjunctions could be represented. The difference this makes is quite remarkable, e.g. for $n=6$ the number of necessary splitting operations increases by a factor of six from 20 to 120.
The impact the most-constrained variable heuristic has on the number of splittings in the global-to-local CPD case (the numbers in brackets in that column) is negligible up to \(n=7\). For \(n=8\) 32 splitting operations are saved (about 8%). The computation time for \(n=7\) was almost unchanged, but for \(n=8\) applying the heuristic led to an increase in computation time from the reported 132.4 to 161.2 seconds.

In an additional trial run, which is not reported in the table, only the first solution was searched for \(n=12\). In the one-rule version, it took 2,816 splitting operations to find it in 75 seconds without CPD. Adding global-to-local CPD reduced the number of splittings to 59 and the processing time to 42.5 seconds. Using global-to-local CPD and the most-constrained variable heuristic reduced the number of splittings further to 51 (while the computation time went up by one second). The results may suggest that global-to-local CPD on its own is more efficient than combined with the heuristic, but this may be due to the nature of the implementation.

In general, the computation times reported here show that the two implementations cannot yet compete with black-box or glass-box constraint solvers in CLP — for example, [VHSaDe93] report a computation time of 0.65 seconds to solve the 8-queens problem in cc(FD) (cf. sections 4.3.2 and 5.1.4.4). Highly specialized C programs can even find all solutions to the 12-queens problem in a few seconds.

### 6.2.3 Map colouring

In the map colouring problem a map, which is partitioned into \(n\) areas, has to be coloured using \(m\) different colours such that no two adjacent areas get the same colour. This problem can easily be stated as a CSP: \(n\) variables \(A_1, \ldots, A_n\) representing the \(n\) areas on the map have to be assigned a value in the domain \(\{c_1, \ldots, c_m\}\), where \(c_1, \ldots, c_m\) denote the different colours, and the assignments have to satisfy the constraint that no two adjacent areas receive the same colour index.
For example, suppose a map of Europe is to be coloured in blue, red, yellow and green. The map can be represented by if-and-only-if definitions:

\[
T_u : \begin{align*}
\text{border(germany, france)} & \leftrightarrow \text{true} \\
\text{border(germany, belgium)} & \leftrightarrow \text{true}
\end{align*}
\]

... 

The constraint that adjacent areas (i.e. countries with a common border) must not receive the same colour can be expressed by an integrity constraint:

\[
IC : \begin{align*}
\text{colour(X, Col)} \land \text{colour(Y, Col)} \land \text{border(X, Y)} & \rightarrow \text{false}
\end{align*}
\]

where colour is an external predicate whose definition represents a colouring of the map.

The goal (obtained from an initial goal by unfolding some user-defined auxiliary predicates) is

\[
\begin{align*}
( \text{colour(germany, blue)} & \lor \ldots \lor \text{colour(germany, green)} ) \land \\
( \text{colour(france, blue)} & \lor \ldots \lor \text{colour(france, green)} ) \land \\
\ldots
\end{align*}
\]

An answer to this goal has to satisfy the integrity constraint and thus represents a colouring of the map such that no two countries with a common border are coloured in the same colour. The complete colouring of the map is returned as a conjunction of colour atoms.

Applying global-to-local propagation is straightforward and effective as it rules out disjuncts corresponding to colourings which are no longer possible. Thus the search tree is pruned in a similar way to the n-queens problem.

However, contrary to the n-queens problem, it is not possible to apply local-to-global CPD. This can be better seen when transforming the single integrity constraint used above into two integrity constraints in a way analogous to the two-rule version for the n-queens problem presented in section 5.1.3:

\[
\begin{align*}
\text{colour(X, Col)} \land \text{border(X, Y)} & \rightarrow \text{notcolour(Y, Col)} \\
\text{colour(Y, Col)} \land \text{notcolour(Y, Col)} & \rightarrow \text{false}
\end{align*}
\]

Here border takes the place of move and notcolour takes the place of attacked in the n-queens problem. In the n-queens problem, different row assignments...
to a queen in one column could lead to the same places on the board being attacked, e.g. both in the queen(1,1) and the queen(1,3) disjunct the atom attacked(1,2) can be generated by propagation. In the map colouring problem, different colour assignments (in different disjuncts of one disjunction) never yield the same notcolour atoms (and thus no atoms can be factored out of any disjuncts).

[RuNo93] propose to use as a heuristic for choosing the next area to be coloured the most-constrained variable strategy. They give an example (a six-area map to be coloured in three colours) where this technique yields a solution without any backtracking once two areas have been coloured. A closer look at the example reveals that the "choice" which is made by the heuristic in each step consists simply in assigning a value to a variable for which there is only one value left anyway. In the CALOG approach this corresponds to a disjunction having been reduced to a single disjunct by global-to-local (and possibly also local-to-local) CPD. The contents of this disjunct are then immediately promoted to global constraints so that no choice is necessary anymore. Thus, in this example, global-to-local CPD has the same effect as the heuristic.

McCarthy [MC81] uses the map colouring problem as an example to evaluate the general practicability of logic programming and of Kowalski's [Ko79a] "doctrine"

Algorithm = Logic + Control.

First he gives a simple example of a six-area map to be coloured in four colours. He points out that a PROLOG program (the logic) is able to find all solutions without having to backtrack over wrong choices, provided that the areas are coloured in a particular order (the control). He then expresses doubts that the right ordering could be identified by the program without an ad hoc intervention by the programmer (which would violate Kowalski's equation). As it turns out, PROCALOG with global-to-local CPD finds all 48 solutions (in about two seconds in the PASCAL implementation) to McCarthy's sample map without making any wrong choices. No ad hoc control is needed.
Admittedly, this positive result owes a lot to the simplicity of McCarthy’s example. McCarthy cites some mathematical results which show that a map can always be coloured in four colours without any backtracking over wrong choices if no area on the map has more than three neighbours. Even better, areas with three or fewer neighbours can be recursively removed from the map until every area on the map has at least four neighbours. If an empty map is obtained as the result of this process, then the original map can always be coloured without backtracking by first colouring the areas which were removed last and so on. This algorithm can be approximated by the most-constraining variable heuristic: always choose the area with the largest number of neighbours to be coloured next. Using this general heuristic appears to be a valid way of satisfying Kowalski’s equation.

Finally, McCarthy describes an algorithm which even allows the elimination of areas with four neighbours (leaving only areas with five neighbours on the recursively downsized map). This time some backtracking may be necessary, and what makes the proposal particularly difficult is that the backtracking should follow a certain scheme of “reassigning” values (e.g. if an area had been coloured in red, then it may be possible to compute that the correct choice would have been green). There does not seem to be an obvious way to implement this algorithm, although an external heuristic procedure supplied by the user might be able to implement it (if sufficient information about the search space is passed to it).

6.3 Operations Research applications

Operations Research (OR) applications include, among others, planning, scheduling, time-tabling and sequencing problems. Many of these problems are \( \mathcal{NP} \)-complete (cf. section 5.1.1.1) and thus are often regarded as too hard for standard LP. This was a motivation behind the black-box CLP approach which integrated specialized mathematical problem solving methods into (constraint)
logic programming.

For efficiency purposes, it may be sensible to do the same in the CALOG framework, i.e. to add a black-box constraint solver which handles frequently encountered constraints (like inequalities or systems of linear equations etc.) which are solvable by standard algorithms (such as the Simplex algorithm or Gaussian elimination), and it would not be hard to do so (cf. section 4.3.2).

However, whether or not this is done, formulations of OR problems in the CALOG framework are potentially easier to understand and more flexible as the framework allows the explicit representation of properties of constraints for which no built-in solver exists. Moreover, the use of the CPD techniques seems to be particularly promising for \( NP \) problems since computational effort invested into pruning the often huge search tree is more likely to pay off. For example, Constructive Disjunction, which corresponds to local-to-global propagation (cf. section 5.1.4.5) has been applied successfully to terminal zone aircraft sequencing problems in [Jo92].

6.3.1 Job-shop scheduling

Different approaches to job-shop scheduling — procedural [ApCo91, CaPi89, Ca82, RK76, MuTh63], black-box CLP [VH89, VHSaDe93], glass-box CLP [CaLa94, CaLa96], and theorem-proving approaches [KoToWe94, To94] — have been studied in [We94].

Here only a (perhaps surprisingly) simple solution, which includes use of the CPD methods, will be presented, and the implementation of a branch-and-bound search strategy will be discussed in the context.

6.3.1.1 Problem definition

An \( m \times n \) job-shop scheduling problem involves \( m \) machines and \( n \) jobs. Each job \( i \) consists of a sequence of \( t(i) \) tasks which have to be performed in the given order. Each task has to be executed on a specific machine and the time this takes (the "processing time") is known. No two tasks can be processed on
the same machine at the same time. Let $t_{ij}$ denote the starting time of task $j$ in job $i$ and let $p_{ij}$ denote its processing time.

The information that the tasks of the same job $i$ have to be executed in the given order can be expressed by the conjunction of the atomic constraints

$$t_{ij} + p_{ij} \leq t_{i,j+1}, 1 \leq j < t(i)$$

The information that one machine can only process one task at a time can be represented by a conjunction of disjunctive constraints, one for each pair of tasks that have to be executed on the same machine. If task $j_1$ of job $i_1$ and task $j_2$ of job $i_2$ have to be executed on the same machine, the resulting constraint can be expressed by the disjunction (cf. section 5.1.1.1)

$$(t_{i_1,j_1} + p_{i_1,j_1} \leq t_{i_2,j_2}) \vee (t_{i_2,j_2} + p_{i_2,j_2} \leq t_{i_1,j_1})$$

In addition to the above constraints, the initial goal contains atomic constraints

$$l_{ij} \leq t_{ij} \leq u_{ij} \text{ for } 1 \leq i \leq n, 1 \leq j \leq t(i)$$

where the $l_{ij}$ and $u_{ij}$ are constants specifying pre-computed lower and upper bounds for the task starting times (in the worst case, all $l_{ij}$ can be set to 0 and all $u_{ij}$ to the sum of all processing times).

In a famous 10x10 job-shop scheduling problem from [FiTh61], each job has 10 tasks so that there are $\frac{10 \cdot 9}{2} = 45$ disjunctive constraints for each machine, 450 in total. Constructing the full search space for the problem would yield a search tree with $2^{450}$ nodes (cf. section 5.1.1.1).³

### 6.3.1.2 Branch-and-bound

Like most OR applications job-shop scheduling is an optimization problem which means that not all possible solutions are of interest, but rather one optimal (or near-optimal) solution is to be identified.

In the job-shop scheduling application, the total processing time, $\text{time}$, which is the maximum over the finishing times of all tasks, is to be minimized. Initially,

\[ 2^{450} = (2^{10})^{45} \approx (10^3)^{45} = 10^{135}, \text{ i.e. a "1" followed by 135 "0"s.} \]
\[ l \leq \text{time} \leq u \]

where \( l \) is the maximum of the set \( \{l_{ij} + p_{ij} : 1 \leq i \leq n, 1 \leq j \leq t(i)\} \) and \( u \) is the maximum of the set \( \{u_{ij} + p_{ij} : 1 \leq i \leq n, 1 \leq j \leq t(i)\} \).

When the first solution (possible schedule) which yields a shorter total processing time than the current upper bound is found, the upper bound should be updated. Then, all partial schedules (in which some disjunctive constraints are still undecided) where the lower bound on the total processing time already exceeds the new upper bound can be rejected. This search strategy is called \textit{branch-and-bound}.

It is not easy to represent updates of the upper bound on the total processing time in a branch-and-bound search in a logical way. The PROLOG implementation of PROCALOG allows the user to specify propagation rules which are triggered only when a new solution is found. For example,

\[
\text{time} \geq \text{Lower} \land \text{time} \leq \text{Upper} \rightarrow \text{time} \leq \text{Lower}
\]

can be used to update the upper bound on \text{time} (assuming that the lower bound for the new solution represents the finishing time of the last task in the solution). One might try to rewrite this rule into a logical integrity constraint:

\[
\text{localtime} \geq \text{Lower} \land \text{new\_solution\_found} \land \text{globaltime} \leq \text{Upper} \\
\rightarrow \text{globaltime} \leq \text{Lower}
\]

but then \text{new\_solution\_found} would still have to be defined and there is no obvious logical way of doing this.

To reject partial schedules which cannot be better than the best solution found so far, the following integrity constraint can be used:

\[
\text{Lower} \leq \text{time} \land \text{time} \leq \text{Upper} \land \text{Upper} < \text{Lower} \rightarrow \text{false}
\]

\textbf{6.3.1.3 CALOG solution including CPD}

The integrity constraints

\[
L_1 \leq T_1 \land T_1 + P_1 \leq T_2 \rightarrow L_1 + P_1 \leq T_2
\]
\[ T_1 + P_1 \leq T_2 \land T_2 \leq U_2 \rightarrow T_1 \leq U_2 - P_1 \]
can be applied to update lower and upper bounds on starting times.\(^4\) As special instances of transitivity, they are properties of the definition of \( \leq \). Moreover, they can be used not only for ordinary propagation, but also for global-to-local and local-to-local CPD. They were already implicitly used in section 5.1.1.2 when motivating global-to-local propagation using a scheduling example.

Furthermore,

\[ L \leq T \land T \leq U \land U < L \rightarrow \text{false} \]
which can be derived from transitivity and anti-symmetry, can be used to reject inconsistent schedules. Again, the integrity constraint can also be used for CPD (to reject disjuncts without splitting disjunctions and thus to prune the search tree) and was implicitly used in section 5.1.1.2.

The use of local-to-global CPD in its simple form (see definition 5.1.2) is of no use in job-shop scheduling problems. However, it frequently happens that one disjunct of a disjunctive scheduling constraint contains a lower bound constraint \( l \leq t \) and the other disjunct an upper bound constraint \( t \leq u \) where \( u < l \). In this case, \( t \) cannot be in the open interval \((u, l)\).

The (extended syntax) integrity constraint already proposed in section 5.1.2 can thus be applied:

\[ (L \leq T \lor T \leq U) \land U < L \rightarrow T \not\in (U, L) \]
The information thus propagated can then be used to further tighten lower and upper bounds by means of the following two integrity constraints:

\[ L \leq T \land T \not\in (A, B) \land A < L \rightarrow B \leq T \]
\[ T \leq U \land T \not\in (A, B) \land U < B \rightarrow T \leq A \]
which just say that if the lower or upper bound falls into the excluded interval, then the bound can be updated to the respective end of the interval.

Integrity constraints for deletion (cf. section 5.4) can be added to remove

\(^4\)Note the slight abuse of notation: \( T_1 + P_1 \) is supposed to be an expression, but \( L_1 + P_1 \) is supposed to be evaluated. This coincides with the suspension of plus atoms, cf. example 2.2.5.
redundant bounds on starting and finishing times:

\[ L_1 \leq T \land L_1 > L_2 \rightarrow L_2 \leq T \]
\[ T \leq U_1 \land U_1 < U_2 \rightarrow T \leq U_2 \]

Again, these rules are just special instances of transitivity and thus logically unproblematic deletion rules.

Deletion rules can also be specified to remove redundant instances of "\( \notin \)",
e.g.

\[ L \leq T \land B \leq L \rightarrow T \notin (A, B) \]

states that, if the lower bound of \( T \) is \( L \) and \( L \) exceeds the upper bound of the interval \((A, B)\), then the information that \( T \)'s value is not in this interval has become useless (but is still correct, of course).

Computational results for job-shop scheduling problems using the PROLOG prototype have already been reported in [We94]. A problem with six jobs, six machines and six tasks per job, taken from [FiTh61], was solved\(^5\) in 38 seconds after 139 splitting operations if the optimal solution was given as the initial upper bound on the total time. When increasing the initial upper bound by about 20 % (from 55 to 65), the problem was solved in about three minutes after 679 splitting operations. In both cases the most efficient method turned out to be local-to-global CPD in its strong (non-simple) form. Adding global-to-local and local-to-local CPD reduced the necessary number of splitting operations to 35 and 100, respectively, but led to an increase of computation time to two and five minutes, respectively. This may be due largely to the nature of the prototype.

Integrity constraints implementing the task interval approach of Caseau and Laburthe [CaLa94, CaLa96] have also been tried and are listed in [We94]. They are considerably more complicated than the simple adaptations of transitivity and anti-symmetry given above and it seems rather unlikely that they can be used to improve efficiency, although it might be worth trying this with the faster PASCAL implementation as soon as it has been optimized and made

\(^5\)I.e. an optimal schedule was found and proven optimal.
more reliable.

6.3.2 Warehouse location problem

This section describes a simple (but nonetheless $\mathcal{NP}$-complete) type of warehouse location problem. The problem is also used in [VHCa88] to compare different approaches: linear (mathematical) programming, a specialized procedural solution implemented in PASCAL, and a black-box CLP solution in CHIP. After defining the problem and briefly discussing alternative solutions, a CALOG approach will be proposed.

6.3.2.1 Problem definition

Given are $m$ warehouses and $n$ customers to whom goods have to be delivered from the warehouses. The delivery costs depend on the distances from the customers to the warehouses. Moreover, there is a fixed cost for keeping a warehouse open.

In the simple version of the problem it is assumed that every warehouse is able to satisfy the needs of all customers (i.e. there are no capacity constraints on the warehouses), and consequently every customer will be served by the closest warehouse which is open.

The problem is thus reduced to deciding which warehouses should be kept open and which should be closed. As any subset of the warehouses is a candidate solution, there are $2^m$ possible combinations.

The following notations will be used to refer to the problem data:

- $f_i$, $1 \leq i \leq m$, denotes the fixed cost to build warehouse $i$
- $v_{ij}$, $1 \leq i \leq m$, $1 \leq j \leq n$, denotes the cost of transporting all required units from warehouse $i$ to customer $j$
6.3.2.2 Previous approaches

The problem can be easily stated as an Integer Programming problem solvable by standard algorithms. This requires the introduction of $m$ auxiliary variables $w_i$ for the warehouses, with $w_i = 0$ denoting that warehouse $i$ is closed and $w_i = 1$ denoting that warehouse $i$ is open, and the introduction of $n \cdot m$ variables $g_{ij}$, with $g_{ij} = 0$ denoting that warehouse $i$ does not deliver goods to customer $j$ and $g_{ij} = 1$ again denoting the opposite. The total cost, which is to be minimized, can then be written as

$$\sum_{i=1}^{m} \sum_{j=1}^{n} v_{ij} g_{ij} + \sum_{i=1}^{m} f_i w_i$$

which is simply the total transportation cost plus the total fixed cost.

This solution requires very little programming effort, but it is potentially inefficient, as it has to solve a system of $n \cdot m + m$ 0-1 variables and thus explores a search space of $2^{n \cdot m + m}$ nodes, whereas, as has already been mentioned, $2^m$ nodes would be sufficient.

[VHCa88] mention a project to implement a specialized program in PASCAL to solve the problem more efficiently. This resulted in the generation of 2,000 lines of code over a rather long development time. The program was efficient but inflexible (e.g. there was no way to add additional capacity constraints without rewriting significant parts of the program) and impossible to prove correct.

[VHCa88] therefore propose a CLP approach in CHIP. They report that the CHIP program is much shorter (only two pages of code) and that writing it took only a fraction of the time to develop the PASCAL program. It is more efficient than the integer programming solution, but about five times slower than the specialized program.

The CHIP program first initializes the domains of several variables and then generates further constraints on these variables. The variables used include 0-1 variables $w_i$ for each warehouse; variables $g_j$ to denote the warehouse which delivers goods to customer $j$; variables $v_j$ to denote the (actual) transportation
cost for customer $j$ (the value of $v_j$ has to be one of the $v_{ij}$).

There is a constraint between $v_j$ and $g_j$:

\[
\text{element}( g_j, [ v_{1j}, \ldots, v_{mj} ], v_j )
\]

where $\text{element}(\text{Index, List, Element})$ holds iff $\text{Index}$ gives the index of element $\text{Element}$ in the list $\text{List}$. So here $g_j$, which is a number from 1 to $m$, gives the index in the list of transport costs which $v_j$ can take.

When a value of 0 or 1 is assigned to a $w_i$, the domains of $g_j$ and $v_j$ can be further constrained — if $w_i = 0$, then no $g_j$ can be assigned the value $i$ anymore; if $w_i = 1$ and $i$ is the cheapest remaining (not closed) warehouse for customer $j$, then $v_j$ can be assigned the value $v_{ij}$. The domains of the respective other variables are automatically constrained via the $\text{element}$ constraints.

CHIP's built-in constraint solver takes care of the details, but some more code is required to assist it and the actual operational behaviour achieved is obvious only to a user who has detailed knowledge of the workings of the constraint solver (as the authors of [VHCa88] certainly have). Otherwise it is not clear why the chosen representation of the constraints should be particularly suited to solve the problem.

The CHIP program is also already rather far away from the very concise problem specification of the integer programming approach. A language which overcomes some of the disadvantages of the CHIP approach is 2LP [MATr96], which combines linear programming and logic programming (hence the name). Basically, 2LP allows stating auxiliary constraints similar to the ones used in the CHIP program to prune the search tree of the pure integer programming solution. The representation of the constraints in 2LP is closer to the integer programming approach than in CHIP.

6.3.2.3 CALOG solution

While the solution in CHIP is declarative, the problem is not specified very naturally. Rather, the constraints must be stated in such a way that the constraint solver is able to behave as desired.
In the CALOG approach to be presented in this section, the choice to open or close warehouses is represented by the disjunctions (or disjunctive constraints)

\[ \text{open}(i) \lor \text{close}(i), 1 \leq i \leq m \]

The initial goal must be unfolded to the conjunction of all these disjunctions. This part is straightforward and not very interesting. The actual problem solution is provided by the integrity constraints.

But first the total cost has to be represented. As the integer programming solution has shown, the total cost is the sum of the total fixed costs (which depend on which warehouses are open) and the total transportation costs. For the latter a minimum can be given for each customer: the cost associated with the warehouse closest to the customer. Let \( \text{mincost} \) be the sum of all these minima and let

\[ \text{cost}([], []) \geq \text{mincost} \]

denote that the total cost is at least \( \text{mincost} \) when no warehouses are decided to stay open (first empty list as parameter of \( \text{cost} \)) and no warehouses are decided to be closed (second empty list).

As soon as it is decided that a warehouse stays open, its fixed cost has to be added to the total cost and the warehouse has to be added to the list of open warehouses.

\[ \text{open}(W) \land \text{cost}([\text{OpenList}], [\text{ClosedList}]) \geq \text{Cost} \rightarrow \]

\[ \text{cost}([W|\text{OpenList}], \text{ClosedList}) \geq \text{Cost} + f_w \]

One could check in the condition of the integrity constraint whether \( W \) already belongs to \( \text{OpenList} \) and not apply the rule in this case. However, this is not necessary to make the rule logical, and it is not difficult to control the computation in such a way that it never applies this integrity constraint twice for the same warehouse \( W \), cf. the discussion of adding deletion rules below.

If the decision is made to close warehouse \( i \), then the minimum transportation cost increases for all those customers for whom warehouse \( i \) was the cheapest warehouse which had not yet been closed. So for each of those
customers, the cost should be increased by the difference between the trans-
portation cost for warehouse $i$ and the next cheapest one. This information
could be computed based on the problem data, but doing so again and again
would be rather inefficient. Instead the increases in transportation costs can
be represented by an additional predicate $transport$ as follows:

$$transport( Cust, [W|Ws], [C|Cs] )$$

denotes that the increase in total transportation cost for customer $Cust$ upon
closure of warehouse $W$ is $C$ (and $Cs$ is a list of the cost increases for the
warehouses in list $Ws$). Initially, there is one $transport$ fact per customer,
with $W$ being the closest warehouse, $C$ being 0 (since the cost is already part of
$mincost$) and $Ws$ being the list of all other warehouses in increasing order of
transportation costs with $Cs$ the respective increases (the increase associated
with the last warehouse could be $\infty$ to signify that it is not possible to close
all warehouses).

So if a warehouse is closed, the following integrity constraint can be used
to propagate updates on the cost and the $transport$ predicate:

$$close(W) \land transport( Cust, [W|Ws], [C|Cs] ) \land$$
$$cost( OList, CList ) \geq Cost \rightarrow$$
$$cost( OList, [(W,C)|CList] ) \geq Cost+C \land$$
$$transport( Cust, Ws, Cs )$$

A tuple, $(W,C)$, is inserted into the list of closed warehouses because the in-
tegrity constraint may be applicable with several different $transport$ atoms
(so different tuples may be inserted into the list for the same warehouse $W$).
The conjunctive conclusion should be seen as a shortcut for two integrity con-
straints with the same condition and one conjunct in the conclusion.\footnote{Alternatively, one could also use one integrity constraint with an atom of a new predicate
as the conclusion which receives all necessary data as parameters and is then defined in terms
of the actual conjunction. Or, in practice, one might as well allow integrity constraints with
conjunctions in the conclusion.}

This is all that is required to solve the problem (using a branch-and-bound
search similar to the one used for job-shop scheduling) — just two integrity
constraints (or three, depending on how one counts) and the precomputation of the transport facts. In contrast to the CHIP or 2LP solutions, the computational behaviour is completely transparent: decisions are made which warehouses are to stay open and which are closed (by splitting disjunctions), then the effects are propagated using the integrity constraints.

The solution can be improved by adding further integrity constraints for both propagation and deletion (cf. section 5.4).

Firstly, if a transport atom is generated which has only one warehouse left in the list, then the warehouse has to stay open:

\[ \text{transport( Cust, [W], [C] ) } \rightarrow \text{open(W)} \]

Note that the cost is not increased because it has already been taken into account. If an open atom is generated by propagation, then it might contradict a close atom and this should be made explicit:

\[ \text{open(W) \land close(W) } \rightarrow \text{false} \]

These two integrity constraints together can also be used for global-to-local propagation to decide disjunctions without splitting.

To further increase efficiency, deletion rules should be provided. Obvious candidates for redundant information to be deleted are the cost inequalities. A simple version of transitivity is not sufficient here as the parameters of cost keep changing. The following integrity constraints can be used instead (as deletion rules):

\[ \text{cost([W\mid OList],CList)\geq NewCost} \]
\[ \rightarrow \text{cost(OList,CList)\geq NewCost-}_W \]
\[ \text{cost(OList,[(W,C)\mid CList])\geq NewCost} \]
\[ \rightarrow \text{cost(OList,CList)\geq NewCost-C} \]

Moreover, transport atoms also get outdated:

\[ \text{transport( Cust, Ws, Cs )} \]
\[ \rightarrow \text{transport( Cust, [W\mid Ws], [C\mid Cs] )} \]

The last deletion rule is not strictly logical, i.e. it is not a property of the underlying theory, because W and C in the conclusion atom could take arbitrary
values (yielding incorrect results if the rule was used for propagation instead of deletion). This can be rectified by either computing the correct values of W and C based on the problem data (this would be computationally expensive) or by storing the values of W and C in additional parameters of transport (when propagating with the integrity constraint for closing a warehouse). As long as the rule is used only for deletion, these changes have no effect, however, and the first two deletion rules could also be simplified (but thus made illogical) by omitting the computation of the actual old cost in the conclusion (i.e. \texttt{NewCost-f}_W and \texttt{NewCost-C} could simply be replaced by a dummy variable \texttt{OldCost}).

Use of the deletion rules not only increases efficiency, but also makes the generated answers easier to interpret. For example, without the deletion rules for the cost inequalities, the first integrity constraint (which deals with the case that a warehouse is to stay open) might generate further inequalities with duplicate occurrences of the open warehouse in the first parameter of cost. The fixed cost would be counted repeatedly, and in the end it would be necessary to identify the one cost inequality in which all open warehouses occur only once.

6.3.2.4 Use of CPD

There are at least three possible ways of applying CPD in the warehouse location problem.

- Global-to-local CPD may be able to decide a disjunction by propagating \texttt{false} into one of the disjuncts.

- Local-to-local CPD may be used to update local cost inequalities.

- Local-to-global CPD could be used to propagate the minimum cost increase caused by either opening or closing a warehouse out of a disjunction before it is decided.
Global-to-local propagation may reject disjuncts using the integrity constraint which states that a warehouse cannot be both open and closed. Global-to-local propagation can also be performed using a global upper bound on the total cost
\[ \text{totalcost} \leq u \]
in a branch-and-bound search (cf. section 6.3.1.2) to reject disjuncts which would lead to a cost higher than the upper bound on the total cost (the local costs having been computed by local-to-local propagation).

Local-to-global propagation is also (and only) useful in a branch-and-bound search: the minimum of the possible local costs (i.e. of the costs incurred by either keeping a particular warehouse open or closing it) can be propagated out of the disjunction:
\[
\text{cost}([W|01], C1) \geq \text{Cost1} \lor \text{cost}(02, [(W, C|C2)]) \geq \text{Cost2} \\
\rightarrow \text{cost}(W) \geq \min(\text{Cost1}, \text{Cost2})
\]
The information provided by the new \( \text{cost}(W) \) inequalities should be combined so that a violation of the upper bound can be detected early. Unfortunately, the administrative costs involved in such computations appear to be quite high. Use of an aggregation operator as discussed in section 6.4.2 might help (but has not yet been implemented).

6.3.2.5 Computational results

The CALOG solution has been tested on a number of randomly generated\(^7\) problem instances with up to 12 warehouses. The following results are taken from [We95] where the integrity constraints for propagation and deletion given in section 6.3.2.3 were first implemented (with some negligible differences) and executed using the PROLOG implementation. As the PASCAL implementation still has problems with recursive integrity constraints and as it cannot yet handle deletion rules, only the results for the PROLOG implementation are reported.

\(^7\)Random within certain sensible parameters.
The first column in the table below gives the number of warehouses times
the number of customers. Recall that the number of warehouses determines
the size of the search space. The second column gives the number of logical
inferences, the computation time in seconds and the number of splitting op-
erations when applying only ordinary propagation (i.e. global-to-global). The
last column gives the same information when global-to-local CPD is added.

<table>
<thead>
<tr>
<th>Wh. x cust.</th>
<th>ord. propagation</th>
<th>global-to-local CPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 x 40</td>
<td>35,072 2.3 15</td>
<td>87,865 5.3 2</td>
</tr>
<tr>
<td>8 x 25</td>
<td>100,465 6.2 66</td>
<td>175,139 11.0 10</td>
</tr>
<tr>
<td>10 x 20</td>
<td>472,839 30.8 360</td>
<td>2,663,844 238.0 124</td>
</tr>
<tr>
<td>12 x 30</td>
<td>865,451 67.0 605</td>
<td>7,560,288 820.0 129</td>
</tr>
</tbody>
</table>

The table shows that, as in the n-queens problem (cf. section 6.2.2), the size
of the search tree is reduced significantly by applying global-to-local CPD. On
average, the number of splitting operations is reduced to about one fifth
(the results in the n-queens problem were even more impressive, see the table
there). Computation times increase due to the inefficiency of the PROLOG
prototype. The equally notable increase in the number of inferences suggests
that the program is doing a lot of unnecessary computations which, as the
results obtained for the n-queens problem indicate, may be avoided by the
PASCAL implementation.

Only the logical integrity constraints from section 6.3.2.3 were used in ob-
taining the above results. The same problems were then also tried with a set
of simpler, but non-logical propagation and deletion rules. In these rules, the
parameters to the cost inequality are omitted, i.e.

\[ \text{cost} \geq \text{C} \]

is used to denote the lower bound on the total cost regardless of which ware-
houses have been decided already. Similarly, the simple, non-logical form of
the deletion rule for the transport predicate given in section 6.3.2.3 can be
used.
The non-logical rules yield the same solutions as the logical ones because their operational behaviour is equivalent under the given implementation of the proof procedure described in section 6.1, including the convention that deletion rules are applied whenever and as soon as possible. The non-logical rules would generate wrong solutions if a different strategy were implemented.

The computational advantage of the non-logical rules is quite significant, e.g. for the 12x30 problem, the time decreases from 67.0 to 42.5 seconds, a saving of more than 20%. At the same time, the number of inferences decreases only by about 10% (while the number of splittings stays the same, of course).

[VHCa88] report that their CHIP program was able to solve a 21x20 problem in 20 seconds. The PROLOG prototype interpreter is not efficient enough to deal with problems of this size, although it was able to find some solutions in a couple of minutes (before it ran out of memory). Finding better initial upper bounds on the total costs and incorporating one of the heuristics discussed in section 6.2.1 might help. Also, given the speed-up which the PASCAL implementation provided in the n-queens problem, it may turn out that PROCALOG is particularly well-suited for representing and solving warehouse location problems.

6.4 Configuration problems

Example 2.2.9 sketched a simple computer configuration problem. In practice such problems can be much more difficult, and specialized software has been developed to deal with them. R1/XCON [MD82] was the first large-scale expert system to be commercially used for configuring computer systems. It is based on more than 10,000 condition-action rules which, unfortunately, have no obvious declarative semantics.

A logical approach to configuration based on semantic networks can be found in [SeNo90]. Other systems have meanwhile been developed which are more closely related to the CALOG framework's way of programming.
RaPiD ("AI in removable partial denture") [HaDaFi93] uses logical integrity constraints to configure dentures in prosthetic dentistry. For example, the information that a rest can only be placed on a tooth which is present (rather than missing) can be represented by the integrity constraint
\[
\text{rest.on.tooth}(\text{Rest}, \text{Tooth}) \rightarrow \text{present}(\text{Tooth})
\]
The denial
\[
\text{present}(\text{Tooth}) \land \text{missing}(\text{Tooth}) \rightarrow \text{false}
\]
then detects and rejects impossible configurations of rests.

Perhaps most closely related to the CALOG framework is the Constructive Problem Solving (CPS) approach of [KlBuNu94]. The following sections analyse the CPS approach and discuss how its main features can be translated into the CALOG framework.

### 6.4.1 Semantics for the CPS approach

[KlBuNu94] propose to view a configuration problem as the task of constructing a model for the domain of devices to be configured.

More precisely, let \( \Sigma \) be the signature of a logical language and let \( \Sigma_0 \) be a sub-signature of \( \Sigma \) which gives the vocabulary to name the devices to be configured and the relations between them.

Given an initial goal \( G_0 \), a theory \( T \) of definitional knowledge (see next section) over \( \Sigma \) and a set \( IC \) of (integrity) constraints, \( C \) constitutes a solution to \( G_0 \) if

1. \( C \) is a finite \( \Sigma_0 \)-structure
2. \( C \models_T \exists G_0 \)
3. \( C \models_T IC \)

where \( C \models_T F \) expresses that \( F \) holds in the (uniquely defined, cf. [KlBuNu94]) extension of \( C \) in \( T \).
Ignoring some technical details, this specification can be translated into CALOG terminology as follows: $T$ corresponds to a theory $\mathcal{T}$ of all definitions (including the configured devices), $C$ is a model of a subtheory of abducibles and thus corresponds to a (candidate) theory $\mathcal{T}_c$ of the external predicates. Constructing a model for the devices to be configured is analogous to discovering possible definitions for the external predicates.

The "$\models_\mathcal{T}$" in (2) and (3) can be regarded as an instance of the propertyhood view. Given that $C$ corresponds to a part of $\mathcal{T}$, (2) could be rewritten as

$$\mathcal{T} \models_{int} \exists G_0$$

which is a consequence of

$$\mathcal{T} \models_{int} \forall [A \rightarrow G_0] \text{ and } \mathcal{T} \models_{int} \exists A$$

which together represent the Horn clause case of conditions (A2) and (A3) in the CALOG semantics (where the only intended model is the minimal Herbrand model) and where $A$ can be regarded as the conjunction of all external atoms which are true in the model $C$ (which, by condition (1), is finite).

Condition (3) then translates into the usual propertyhood view of integrity constraint satisfaction,

$$\mathcal{T} \models_{int} IC$$

These similarities between a semantics defined for a specific approach to one class of applications and the CALOG semantics once again illustrate the wide applicability of the unifying principles on which the CALOG framework is based.

6.4.2 Knowledge representation in the CPS approach

In the CPS approach, the definitional knowledge $T$ is defined in terms of taxonomic hierarchies which are typical for terminological reasoning systems like KL-ONE [BrSc85]. [Fu93] gives examples how terminological reasoning can be performed in a normal LP framework enhanced by integrity constraints and suggests that such a translation has conceptual advantages. [FrHa94] shows
how CHRs, which correspond to integrity constraints, can be used to simulate terminological reasoning.

The different ways of representing knowledge in the CPS approach of [Kl-BuNu94] will now be presented by means of examples and a translation into the CALOG framework is proposed in each case.

Cover axioms define the taxonomic hierarchies. They are of the form

\[ A := A_1 \lor \ldots \lor A_n \]

It is clear from [Kl-BuNu94] that cover axioms are meant as if-and-only-if definitions: an object is of type \( A \) if it is of some subtype \( A_i \), and only objects of subtypes \( A_1, \ldots, A_n \) are of type \( A \). For example (taken from a supplement to [Kl-BuNu94]):

\[
\begin{align*}
\text{fruit} & := \text{apple} \lor \text{banana} \lor \text{cherry} \\
\text{apple} & := \text{granny} \lor \text{delicious} \\
\text{box} & := \text{small\_box} \lor \text{large\_box} \\
\text{state} & := \text{germany} \lor \text{france}
\end{align*}
\]

Translated into if-and-only-if definitions:

\[
\begin{align*}
\text{fruit}(X) & \leftrightarrow \text{apple}(X) \lor \text{banana}(X) \lor \text{cherry}(X) \\
\text{apple}(X) & \leftrightarrow \text{granny}(X) \lor \text{delicious}(X)
\end{align*}
\]

and so on.

Inclusion axioms specify properties of objects of certain types. They are of the form

\[ A :< E \]

where \( A \) is a type and \( E \) is a restricted sort expression which objects of type \( A \) have to satisfy. For example:

\[
\begin{align*}
\text{fruit} :< \text{placed: box, from: state}
\end{align*}
\]

If something is a fruit, then it must be placed in something that is a box and it must be from something that is a state. This can be translated into an implication

\[
\begin{align*}
\text{fruit}(X) & \rightarrow \exists B [ \text{placed}(X,B) \land \text{box}(B) ] \land \\
& \exists S [ \text{from}(X,S) \land \text{state}(S) ]
\end{align*}
\]

213
The implication is not in the proper form of an integrity constraint due to the existential quantifiers and conjunctions in the conclusion. But it is trivial to fix this: simply use the integrity constraint

\[ \text{fruit}(X) \rightarrow \text{box-and-state}(X) \]

and add the definition

\[ \text{box-and-state}(X) \leftrightarrow \text{placed}(X,B) \land \text{box}(B) \land \text{from}(X,S) \land \text{state}(S) \]

where the existential quantifiers are now implicit. The original implication is (operationally) equivalent to the integrity constraint together with the if-and-only-if definition.

Some inclusion axioms are more appropriately represented by definitions than by integrity constraints, e.g.

- apple :< weight: 10
- banana :< weight: 25
- cherry :< weight: 25

easily translates into a definition for a predicate weight:

\[ \text{weight}(X,Y) \leftrightarrow \left[ \text{apple}(X) \land Y=10 \right] \lor \left[ \text{banana}(X) \land Y=25 \right] \lor \left[ \text{cherry}(X) \land Y=5 \right] \]

Disjointness axioms prohibit objects to be of different, incompatible types at the same time. They are of the form

\[ A_1 \parallel || A_2 \]

For example, something cannot be both an apple and a banana

\[ \text{apple} || \text{banana} \]

etc. Disjointness axioms can be translated into denial integrity constraints

\[ \text{apple}(X) \land \text{banana}(X) \rightarrow \text{false} \]

etc. As the cover axioms are formulated as if-and-only-if definitions, all types that are not sub- or supertypes of each other (in the usual sense within the taxonomic hierarchy) should be disjoint.

Constraints restrict possible configurations. They are of the form
Conditions → Conclusion

For example, the constraints

\[
\begin{align*}
X: \text{apple}, \ X.\text{from}=Y & \rightarrow Y: \text{germany} \\
X: \text{banana}, \ Y: \text{france}, \ X.\text{from}=Y & \rightarrow \text{false}
\end{align*}
\]

state that apples always come from Germany and bananas never come from France. They can be represented as integrity constraints

\[
\begin{align*}
\text{apple}(X) \land \text{from}(X,Y) & \rightarrow Y=\text{germany} \\
\text{banana}(X) \land \text{from}(X,\text{france}) & \rightarrow \text{false}
\end{align*}
\]

Other constraints used in [KIBuNu94] make use of special cardinality and aggregation operators, e.g. the constraints

\[
\begin{align*}
B: \text{box}, \ ((F.\text{card}(F: \text{fruit}, F.\text{placed}=B)) > 2) & \rightarrow \text{false} \\
B: \text{box}, \ ((F.\text{weight}\sum(F: \text{fruit}, F.\text{placed}))) > 30 & \rightarrow \text{false}
\end{align*}
\]

express that at most two fruits can be placed in one box and that the sum of the weights of the fruits in a box must not exceed 30. These constraints could be translated into:

\[
\begin{align*}
\text{box}(B) \land \text{card}(F,\text{placed}(F,B),C) \land C>2 & \rightarrow \text{false} \\
\text{box}(B) \land \text{sum}(W, (\text{placed}(F,B),\text{weight}(F,W)),S) \land S>30 & \rightarrow \text{false}
\end{align*}
\]

where card and sum are meta-logical predicates which can be defined, either by the user or built-in, using predicates like PROLOG’s findall. Such meta-logical predicates are useful in several situations and should probably be included in the implementation (the PROLOG prototype can do this already by using findall).

The last means of knowledge representation in the CPS approach are relations which define ordinary predicates in clausal form:

\[ \text{Head} \leftarrow \text{Body} \]

For example, the relation well.placed may be defined by

\[
\begin{align*}
\text{well.placed}(X,Y) \leftarrow X: \text{apple}, \ Y: \text{large.box}, \ X.\text{placed}=Y \\
\text{well.placed}(X,Y) \leftarrow X: \text{cherry}, \ Y: \text{small.box}, \ X.\text{placed}=Y \\
\text{well.placed}(X,Y) \leftarrow X: \text{banana}, \ Y: \text{box}, \ X.\text{placed}=Y
\end{align*}
\]

which can easily be translated into one if-and-only-if definition:
6.4.3 Executing a goal

Given the axioms, constraints and relations from the previous section, the goal
A: apple, B: box, well_placed(A, B)
has the solution
A: granny, A. placed=B, B: large box, C: germany, A. from=C
and an analogous one with A: delicious. Due to the different kinds of knowl-
edge representation, it is not obvious how these solutions are computed. In
fact, [KlBuNu94] define a total of 12 operations to transform a goal and com-
pute a solution. The solution is returned in the form of a model (cf. sec-
tion 6.4.1) which is initially empty and constructed by the operations.

For example, one operation removes an atom from the goal, stores it in
the model and adds all applicable (“induced”; see the precise definition in
[KlBuNu94]) inclusion axioms and constraints to the goal. For the given goal,
when A: granny (respectively, A: delicious) is moved to the model, then
A. placed=B, C: state, A. from=C, C: germany
is added to the goal.

Now consider the execution of an initial goal corresponding to the above
goal in the CALOG framework.
apple(A) ∧ box(B) ∧ well_placed(A, B)
Unfolding well_placed and propagating with the integrity constraints which
represent the disjointness axioms (an apple cannot be a cherry and neither
a banana, so only one of the disjuncts which result from unfolding survives)
yields:
apple(A) ∧ box(B) ∧ placed(A, B) ∧ large_box(B)
Using the implications which represent the inclusion axioms\footnote{Note that the condition of the integrity constraint} for propagation
adds (after unfolding)

\[ \text{placed}(A, B') \land \text{box}(B') \land \text{from}(A, C) \land \text{state}(C) \]

Now propagation can be applied using the integrity constraint which states that apples always come from Germany. It adds the equality \( C=\text{germany} \) to the goal (which can be used for some equality rewriting).

The goal still contains several reducible atoms and some more steps of unfolding and propagation follow. An interesting aspect is the presence of two variables, \( B \) and \( B' \), both representing boxes. The apple is placed in \( B' \), but the intention was to store it in \( B \). This solution is implicit, however, since the case \( B=B' \) is not excluded. If the factoring operation (cf. section 4.2.2.6) was added to the proof procedure, then two solutions, one with the apple in box \( B \) and one in a box \( B' \) different from \( B \) would be generated. [KIBuNu94] define an operation \textit{feature unify} which ensures that only the case \( B=B' \) is considered. This can be justified by minimality requirements imposed on the solution (cf. the discussions in section 4.2).

The discussion of the CPS approach has shown that the CALOG framework may provide a better and more general understanding of rather specific ways of representing and solving problems. The CALOG proof procedure’s small kernel of basic operations makes it easier to follow the problem solving process. Even if more operations should be necessary for efficiency purposes, it is generally useful to distinguish between basic operations and problem specific extensions. Representing configuration problems in the CALOG framework also has the advantage of making the relationship to ALP and CLP more obvious. [KIBuNu94] mention ALP only in their concluding discussion and dismiss existing abductive reasoning systems as having no “realistic applications”. Their criticism may even be valid for purely abductive systems, but, as this thesis hopes to demonstrate, the solution is not to abandon abduction altogether but rather to integrate it into a unified framework.

\[ \text{fruit}(I) \rightarrow \text{box_and_state}(I) \]

is reducible and can be unfolded to yield, after normalization, three new implications, one of which has the condition \text{apple}(I) and is thus applicable to the goal.

217
6.5 Logic programming with integrity constraints

Integrity constraints can also be used to improve the efficiency of standard LP programs (which can be translated into the CALOG framework with only minor modifications). In the first example given below, they are added as reasoning shortcuts, in the second example they are used to replace an inefficient definition.

6.5.1 Local propagation for Boolean operators

Consider one possible definition of the Boolean operator and:

\[
\begin{align*}
\text{and}(0,0,0) & \leftrightarrow \text{true} \\
\text{and}(0,1,0) & \leftrightarrow \text{true} \\
\text{and}(1,0,0) & \leftrightarrow \text{true} \\
\text{and}(1,1,1) & \leftrightarrow \text{true}
\end{align*}
\]

Then the atom \(\text{and}(0, Y, Z)\) is suspended. It is indeed not possible to determine the value of \(Y\), but \(Z\) has to be 0 in either of the two ways in which \(\text{and}(0, Y, Z)\) can be reduced. This can be expressed by the integrity constraint

\[\text{and}(0, Y, Z) \rightarrow Z=0\]

Deducing that \(Z\) is 0 may help determine the value of \(Y\) without having to consider the two cases. For example, the constraint

\[Y \leq Z\]

might also be part of the goal and could be used together with the information that \(Y\) has to be either 0 or 1 to conclude that \(Y\) has to be 0.

Similarly, integrity constraints like

\[\text{and}(1, Y, Z) \rightarrow Y=Z\]

and

\[\text{and}(X, Y, 1) \rightarrow (X, Y)=(1,1)\]

may also prove useful. Note, however, that the last one may be more appropriately written as a definition

\[\text{and}(X, Y, 1) \leftrightarrow X=1 \land Y=1\]
which could be used to replace the definition of \(\text{and}(1,1,1)\) given above so that the atom \(\text{and}(X,Y,1)\) is no longer suspended.

Integrity constraints like the above, with single-atom conditions, implement local propagation, see section 4.3.3. Similar integrity constraints could be defined for other Boolean operators.

Parameter dependencies in Boolean operators are also used to illustrate Generalized Propagation in [LPW92]. As has been pointed out in section 5.1.4.5, Generalized Propagation is closely related to Constructive Disjunction [JS93], which itself can be seen as a way of implementing local-to-global CPD (section 5.1.2). For local-to-global CPD to become applicable in the example, the definition of \(\text{and}\) should be written using a disjunction:

\[
\text{and}(X,Y,Z) \leftrightarrow (X,Y,Z) = (0,0,0) \lor (X,Y,Z) = (0,1,0) \lor (X,Y,Z) = (1,0,0) \lor (X,Y,Z) = (1,1,1)
\]

Now the atom \(\text{and}(0,Y,Z)\) from the initial example is not suspended, but can be reduced to a 4-ary disjunction. The third and fourth disjunct are reduced to \textit{false} by equality rewrite rule (EQ2). Further equality rewriting (using rules (EQ1) and (EQ3)) yields the binary disjunction

\[
( Y=0 \land Z=0 ) \lor ( Y=1 \land Z=0 )
\]

The local constraint \(Z=0\) occurs in both disjuncts and can thus be propagated out of the disjunction using local-to-global CPD in its simple form. So the same effect is achieved automatically as when using the four ground definitions of \(\text{and}\) with the explicit integrity constraint \(\text{and}(0,Y,Z) \rightarrow Z=0\).

6.5.2 Computing Fibonacci numbers bottom-up

The series of natural numbers starting with 0,1 in which subsequent series elements are obtained by adding the previous two elements is called the Fibonacci series. The specification can be translated into the if-and-only-if definition

\[
fib(N,F) \leftrightarrow (N=1 \land F=0) \lor (N=2 \land F=1) \lor (N \geq 3 \land fib(N-1,F_{N-1}) \land fib(N-2,F_{N-2}) \land F=F_{N-1}+F_{N-2})
\]
The definition is easy to write as it directly corresponds to the specification. Suppose the initial goal is \( \text{fib}(N,F) \). Then all the terms of the Fibonacci series will be returned as computed answers (infinitely many) in the form of assignments to \( N \) and \( F \), i.e. the goal will be transformed into

\[
(N=1 \land F=0) \lor (N=2 \land F=1) \lor (N=3 \land F=F) \lor \ldots
\]

However, the computation of the Fibonacci numbers by unfolding the definition in conventional LP goal reduction manner is very inefficient and involves the repeated recomputation of instances of \( \text{fib}(N,F) \). The definition can be made more efficient by transforming it into a tail-recursive definition:

\[
\text{fib}(N,F) \leftarrow (N=1 \land F=0) \lor (N=2 \land F=1) \lor (N=3 \land \text{fibO}(N,F,1,1))
\]

\[
\text{fibO}(N,F,\text{Prev},\text{Acc}) \leftarrow (N=3 \land F=\text{Acc}) \lor \text{fibO}(N-1,F,\text{Acc},\text{Acc}+\text{Prev})
\]

Now answers are computed much more efficiently (in linear time) and no instance of \( \text{fib}(N,F) \) is computed more than once.

Unfortunately, the new definition no longer corresponds directly to the specification and is thus less obviously correct. But the efficiency of the tail-recursive definition can also be achieved without having to abandon the specification, using propagation with integrity constraints instead of unfolding of definitions.

The following three integrity constraints are all properties of the definitions of \( \text{fib} \):

\[
\text{true} \rightarrow \text{fib}(1,0)
\]

\[
\text{true} \rightarrow \text{fib}(2,1)
\]

\[
\text{fib}(N,F_1) \land \text{fib}(N+1,F_2) \rightarrow \text{fib}(N+2,F_1+F_2)
\]

These integrity constraints are meant to replace the if-and-only-if definition, not be added to it. Thus \( \text{fib} \) is now treated as an external predicate whose definition is to be discovered. The initial goal is now simply \text{true}, then the
integrity constraints are added to it, and the derivation proceeds by propa-
gating first \( \text{fib}(1,0) \) and \( \text{fib}(2,1) \) into the goal and then, using the third
rule, \( \text{fib}(3,1) \) and so on. The goal thus becomes a conjunction of \( \text{fib} \) atoms
representing the elements of the Fibonacci series.

While the integrity constraints are as efficient as the tail-recursive solution
and much closer to the specification, their use does have disadvantages. For
example, if the value or the number of a particular series element is to be
computed, then the third integrity constraint has to be modified, e.g. by adding
a stopping condition

\[ \ldots \land \text{findfib}(X,Y) \land N<X \land F_1<Y \rightarrow \ldots \]

where \( \text{findfib}(X,Y) \) is the initial goal (with \( X \) and \( Y \) being either variables
or ground terms).

6.6 Other applications

To conclude this chapter some other possible applications from different areas
will be mentioned briefly.

6.6.1 Legal reasoning

[Ko92c] discusses several examples of modelling laws or public notices as logic
programs. The examples include notices on the London Underground such as

\[ \text{Please give up this seat} \]

\[ \text{if an elderly or handicapped person needs it.} \]

This is a condition-action rule and it would require an artificially introduced
purpose to transform it into the form of a logic program, e.g.

\[ \text{You do a good deed} \]

\[ \text{if you give up your seat to a person} \]

\[ \text{who is elderly or handicapped} \]

\[ \text{and who needs your seat} \]

[Ko92c] remarks that the use of integrity constraints would help in directly
modelling requests or commands like the above. Indeed, rather than transforming a condition-action rule into a logic program, in the CALOG framework the original rule could be used as an integrity constraint which can be regarded as a property of somebody who does good deeds. Similarly,

*Do not obstruct the doors*

could be represented by a denial integrity constraint

\[
\text{obstruct\_doors} \rightarrow \text{false}
\]

which is again a property of someone who does good deeds (or behaves well), provided that there is a definition

*You do a good deed when you do not obstruct the doors*

Note that the definitions for doing good deeds (which should be combined into one if-and-only-if definition) can be regarded as inaccessible definitions of external predicates and thus need not be represented directly.

### 6.6.2 The event calculus

The event calculus [KoSe86] and its applications (such as planning) could be represented and studied in the CALOG framework. Interestingly, problems with the original event calculus have led to the definition of a new event calculus which makes use of definitions and integrity constraints in a way similar to the CALOG framework (see [SaKo95]). One of the problems was that certain properties of definitions in the event calculus (which are true in the intended models) may be hard or impossible to prove by using the calculus axioms, e.g. (from [SaKo95]):

\[
\text{start}(\text{Period, Event}_1) \land \text{end}(\text{Period, Event}_2) \rightarrow \text{Event}_1 < \text{Event}_2
\]

Such properties can be made explicit as logical integrity constraints in the CALOG framework.
6.6.3 Reconciling rational and reactive agents

[Ko95] and [KoSa96] propose a unified agent architecture to reconcile rational and reactive agents. The representation of knowledge in the form of both definitions and integrity constraints plays an important role in their approach.

Definitions can be used to model the general beliefs of an agent, i.e. its knowledge about the world.

Integrity constraints can be used to represent both the constraints a rational agent has to obey and the condition-action rules which a reactive agent acts upon, e.g.

\[ \text{do}(\text{Act}, \text{Time}) \land \text{do}(\text{Act'}, \text{Time}) \land \text{Act} \neq \text{Act'} \rightarrow \text{false} \]

is an integrity constraint for a rational agent who cannot do two different things at the same time and

\[ \text{holds}(\text{raining}, \text{Time}) \rightarrow \text{do}(\text{carry-umbrella}, \text{Time}+1) \]

is a condition-action rule for a reactive agent who, in this case, reacts to the information that it is raining by carrying an umbrella.

[Dá97] applies the Iff Proof Procedure of [Fu95] to implement the unified agent architecture. For example, he uses definitions and integrity constraints to control an elevator which has to react to service requests from different floors.
Chapter 7

Conclusion

This thesis has proposed the CALOG framework as a new approach to unify standard Logic Programming, Abductive Logic Programming, Constraint Logic Programming and Semantic Query Optimization. To set up the framework, both new concepts, such as that of the propertyhood view of integrity constraint satisfaction, have been introduced and well-known concepts, such as that of suspension, have been adopted, but given a novel, more general purpose.

A proof procedure has been specified for the framework, based on two previously defined abductive proof procedures [Fu96, DeDS97]. In contrast to its predecessors, the CALOG proof procedure has been defined with generality and simplicity in mind so that it may serve as a template for numerous more practical and more efficient instances, not only for ALP but also for CLP and other areas.

This has been done at the cost of sacrificing stronger soundness and completeness results which are, however, still obtainable for specific instances of the proof procedure. Moreover, the semantics of the CALOG framework has been designed in such a way that it is flexible enough to allow some general soundness results to be proven.

Two implementations of the proof procedure as a programming language called PROCALOG exist and demonstrate the principle feasibility of the ap-
approach — one a prototype programmed in meta-interpreter style in PROLOG, the other a potentially more efficient version in PASCAL. The implementations have been tried on a number of applications, including “real-world” problems from Operations Research. PROCALOG programs are generally simple, natural and flexible. They allow a fast development of prototype solutions which can later be refined or extended (by adding further integrity constraints) without having to completely abandon the original specification (as is often the case in purely procedural approaches).

Further applications could be identified and the advantage of the unifying approach should manifest itself when looking at problems which have no straightforward solution in any one of the framework instances. Traditional applications of Artificial Intelligence like expert systems, or more generally rule-based deduction systems, might be good candidates. The CALOG framework may make a logical approach to non-logical production rule systems possible which preserves more of the structure of the production rule system than the approach taken in [Ra94, RaLo94] (cf. section 5.4.3).

The computational results reported in chapter 6 are no match for more specialized approaches, but nevertheless promising, since they demonstrate the effectiveness of the CPD methods in pruning the search tree and the practicability at least of global-to-local propagation. Definite comparisons are not yet possible due to the preliminary nature of the two available implementations. However, it seems likely that the integration of a black- or glass-box constraint solver for standard tasks like solving systems of equalities and inequalities is necessary to enable PROCALOG to compete with traditional CLP approaches. The treatment of equality (effectively by a glass-box constraint solver implemented by the equality rewriting operation) is a first step in this direction (although it might be interesting, at least for theoretical purposes, to try to identify a “no-box” approach to equality so that a special treatment of the equality predicate is no longer necessary).

Other opportunities for further research have been mentioned at several
points in the thesis. There are two main directions in which this research could lead.

The first is towards a further generalization of the framework which could widen its unifying scope and allow the inclusion of other areas of Logic Programming such as Concurrent Logic Programming and Concurrent Constraint Logic Programming, cf. the discussion in section 4.4. This could also make new classes of applications accessible to the CALOG framework.

The other direction is towards a specialization of the proof procedure which is more efficient and more complete, at least with respect to some instances of the semantics, such as the three-valued completion semantics (which is also used in the related abductive proof procedures). The proof of additional soundness and completeness results both for the general framework and for instances should then become possible.

The aims of achieving stronger theoretical results can be achieved both by adding new operations to the proof procedure and by restricting existing operations, especially propagation. Several possible extensions, like case analysis, factoring, deletion and subsumption, and restrictions, like $P_1$- and hyper-resolution and guarded propagation, have been discussed in chapters 4 and 5 and some of them have already been successfully implemented.

Maybe the most interesting extension of the basic proof procedure is the definition of the CPD methods for propagation across disjunct boundaries. The CPD methods may be regarded as a unifying platform for several specialized approaches to handling disjunctive constraints in CLP (see section 5.1.4).

It may also be worth exploring links between the CALOG framework and theorem-provers like SATCHMO [MaBr88], which could be regarded as a theorem-proving instance of the CALOG framework and its proof procedure. Just like the CALOG framework, SATCHMO lets the user specify integrity constraints (but no definitions). SATCHMO uses them for satisfiability checking and to generate models by forward reasoning (the acronym SATCHMO stands for SATisfiability CHecking by MOdel generation). This is similar to the
treatment of integrity constraints by the CALOG proof procedure and to the interpretation of the computational task as discovering a theory $T_e$ of the external predicates. [MaBr88] report a problem with refutation soundness which they solve by imposing range-restrictedness on integrity constraints (whereas the CALOG proof procedure uses the more general safety condition which permits some non-range-restricted programs). SATCHMO's model generation procedure is essentially propagation by hyper-resolution (cf. section 5.2.2).

To further establish links to theorem-proving, the relationship of the CPD methods to non-clausal resolution [Mu82] could be examined. It might be possible to obtain a more general definition of CPD and also a more elegant definition of local-to-global CPD in its non-simple form.
References


229


230


[Fu93] Fung, T. H.: Theorem Proving Approach with Constraint Han-
Abduction and Its Applications on Databases, M.Sc. Thesis, Imperial College 1993


the 4th Int. Conf. on Artificial Intelligence in Design, 1994


S. (ed.): Proc. of the 10th Int. Conf. on Logic Programming, pp. 737–753, MIT Press 1993


[Ra94] Raschid, L.: A Semantics for a Class of Stratified Production
System Programs, Journal of Logic Programming 21 (1), 1994


[VHSaDe93] Van Hentenryck, P.; Saraswat, V. A.; Deville, Y.: *Design, Imple-


## Glossary of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKL</td>
<td>Andorra Kernel Language</td>
</tr>
<tr>
<td>ALP</td>
<td>Abductive Logic Programming</td>
</tr>
<tr>
<td>CALOG</td>
<td>Constraints and Abduction in LOGic</td>
</tr>
<tr>
<td>cc(FD)</td>
<td>concurrent constraint programming over Finite Domains</td>
</tr>
<tr>
<td>CCLP</td>
<td>Concurrent Constraint Logic Programming</td>
</tr>
<tr>
<td>CET</td>
<td>Clark's Equality Theory</td>
</tr>
<tr>
<td>CHIP</td>
<td>Constraint Handling In Prolog</td>
</tr>
<tr>
<td>CHR</td>
<td>Constraint Handling Rule</td>
</tr>
<tr>
<td>CLP</td>
<td>Constraint Logic Programming</td>
</tr>
<tr>
<td>CPD</td>
<td>Constraint Propagation with Disjunctions</td>
</tr>
<tr>
<td>CPP</td>
<td>CALOG Proof Procedure</td>
</tr>
<tr>
<td>CPS</td>
<td>Constructive Problem Solving</td>
</tr>
<tr>
<td>CSP</td>
<td>Constraint Satisfaction Problem</td>
</tr>
<tr>
<td>EDB</td>
<td>Extensional DataBase</td>
</tr>
<tr>
<td>IC</td>
<td>the set of Integrity Constraints</td>
</tr>
<tr>
<td>IDB</td>
<td>Intensional DataBase</td>
</tr>
<tr>
<td>IPP</td>
<td>Iff Proof Procedure</td>
</tr>
<tr>
<td>LP</td>
<td>Logic Programming</td>
</tr>
<tr>
<td>NP</td>
<td>Non-deterministic Polynomial</td>
</tr>
<tr>
<td>OR</td>
<td>Operations Research</td>
</tr>
<tr>
<td>PROCALOG</td>
<td>PROgramming in CALOG</td>
</tr>
<tr>
<td>SLD</td>
<td>Select-Linear-Definite resolution strategy</td>
</tr>
<tr>
<td>SLDNF</td>
<td>SLD-resolution with Negation as Failure</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>SLDNFA</td>
<td>SLDNF extended by Abduction</td>
</tr>
<tr>
<td>SQO</td>
<td>Semantic Query Optimization</td>
</tr>
<tr>
<td>$\mathcal{T}$</td>
<td>the Theory of if-and-only if definitions</td>
</tr>
<tr>
<td>$\mathcal{T}_b$</td>
<td>built-in part of $\mathcal{T}$</td>
</tr>
<tr>
<td>$\mathcal{T}_e$</td>
<td>external part of $\mathcal{T}$</td>
</tr>
<tr>
<td>$\mathcal{T}_u$</td>
<td>user-defined part of $\mathcal{T}$</td>
</tr>
</tbody>
</table>