Graph Transformation Systems in Constraint Handling Rules: Improved Methods for Program Analysis

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

Frank Raiser
aus Füssen

Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Programmiermethodik und Compilerbau
Institutsdirektor: Prof. Dr. Helmuth Partsch

November 2010
Amtierender Dekan: Prof. Dr.-Ing. Klaus Dietmayer

1. Gutachter: Prof. Dr. Dr. Thom Frühwirth (Ulm University)
2. Gutachter: Prof. Dr. Enno Ohlebusch (Ulm University)
3. Gutachter: Prof. Dr. Slim Abdennadher (German University in Cairo)

Tag der Promotion: 09. November 2010
Acknowledgments

As with any work of this size and scope, it only came to existence due to a seemingly random chain of events and the more or less direct influence of other people. At this point, I would like to take some time (or rather space) to express my gratitudes to those who are responsible for this work coming to fruition, probably as much as myself, if not more so.

First and foremost, I want to thank my supervisor Thom Frühwirth. Through working for him, I came to know about the intricacies involved in academic research, of which only very few are related to actual scientific topics. As the years progressed, he supported me in learning the ropes, which proved to be just as important for a doctoral career as the scientific content itself. Last, but not least, he supported my urge to travel around the world and visit different places, especially when there were suitable conferences nearby.

Deepest thanks also go to Marc Meister, who accompanied my journey in the early years and in countless hours of discussions revealed not only tremendous knowledge to me, but also his artistic talent – for his words continuously painted pictures of my future, ranging from the bleak to the bright. Looking back at these pictures from where I stand now, they served me well in avoiding those dark pitfalls.

I also want to thank the whole institute of Software Engineering and Compiler Construction, which became my second home for all these years. In particular, its director Helmut Partsch, who despite his many duties always remained approachable and helpful. And I want to thank my co-workers Ingmar Baetge, Hariolf Betz, Armin Bolz, Marcel Dausend, Ralf Gerlich, Dominik Gessenharter, Walter Guttmann, Carolin Hürster, Jens Kohlmeyer, Sandra Mann, Alexander Raschke, Stefan Sarstedt, and Matthias Schneiderhan for providing a fantastic atmosphere. Although they all perform research in other areas and we spared no excuse to light-heartedly ridicule each other’s, we all became good friends and shared much more than just the burden of work. In my few years here, the institute also saw three secretaries – Claudia Hammer, Ulrike Seiter, and Eva Englert – to whom not only I am grateful, as their work gave all of us more time to persue our doctoral studies.

Further thanks go to the many scientific colleagues I have met all around the world for the numerous fruitful discussions. I’d like to especially thank Jon Sneyers and Peter van Weert with whom it was a pleasure to organize our various meetings and Slim Abdennadher for the wonderful time in Egypt.

Finally, I am grateful to all my friends, and in particular Martina Reng, for supporting me in any way possible during these often chaotic years. Their presence and friendship has truly enriched my life.

In the end, let me return to the beginning and thank my parents who have successfully
prepared me for this journey and unfailingly believed in me. Due to my father’s untimely
death he could no longer witness the completion of this work and because his life was as far
from its subject as one could imagine, I believe it to be even more appropriate to dedicate
this thesis to him.

\textit{to my father}
Summary

In recent years, the importance of rule- and logic-based technologies steadily intensified, and nowadays, we witness their increased industrial adoption. Constraint Handling Rules (CHR) is one of these formalisms and has established itself as an active research topic during the last two decades. In contrast to other rule-based approaches, CHR is both, a theoretical formalism related to first-order and linear-logic, and a practical rule-based programming language.

Research on CHR revealed, that it is a remarkable declarative language. For example, although it is logic-based, every algorithm can be implemented in CHR in optimal time and space complexity. Furthermore, its execution efficiency is outstanding, in that it consistently outperforms other state-of-the-art production rule systems.

Other rule- and logic-based approaches, like term rewriting, logical algorithms, or petri nets, have been successfully embedded in CHR. For this reason, it is considered a candidate for a lingua-franca of such approaches, which provides the basis for our work. We investigate CHR’s suitability for this purpose exemplarily, by considering an embedding of graph transformation systems (GTSs) in CHR, which helps us in identifying points of improvements for strengthening the lingua-franca argument further.

In particular, we identify shortcomings of current formulations of the operational semantics of CHR. We present a novel formulation in order to alleviate these, which is founded on an equivalence relation over CHR states. It justifies the perspective on CHR as a rule-based rewriting system of equivalence classes, which abstracts over infinitely many possible syntactic variations of a CHR state. Furthermore, the resulting transition system is suitable for teaching CHR, as it is given by a single intuitive inference rule. Overall, this equivalence-based operational semantics provides a powerful formal analysis tool for CHR, which can significantly reduce proof complexity.

The lingua-franca argument implies a potential for cross-fertilization of research. Hence, we revisit program analysis methods available in the CHR literature from the point of view of our equivalence-based operational semantics. We extend existing program analysis methods, in particular for confluence and program equivalence, to support program invariants. The traditional methods often fail for states that are irrelevant, given the assumptions made by the programs’ developers. Our extended methods apply an invariant to make these implicit assumptions explicitly available during analysis, hence, effectively eliminating irrelevant states. The resulting methods are applicable to strictly larger classes of CHR programs and provide more promising approaches towards analyzing more complex programs.

We then return to the above lingua-franca argument, by reconsidering our embedding of graph transformation systems in CHR under the equivalence-based operational semantics. This
confirms the suitability of our improvements to the operational semantics, by for example, revealing a closer correspondence between graph isomorphism and state equivalence in CHR. Finally, we put the research cross-fertilization argument of a lingua-franca to the test, when comparing program analysis methods in both systems. Confluence is particularly interesting, because its decidability differs for GTSs and CHR programs. We find that the adapted CHR confluence test, when applied to a GTS embedded in CHR, corresponds to a sufficient criterion for confluence of the GTS. Similarly, we introduce program equivalence for GTSs with a sufficient criterion based on the CHR program equivalence test.

In summary, our work provides novel program analysis methods that are capable of examining programs under the environment assumed by their developers. We exemplarily applied the lingua-franca argument to GTSs, which yielded relevant insights into confluence analysis of CHR programs and a program equivalence test for GTSs. In conclusion, our work provides a stronger foundation for CHR as a lingua-franca, based on the more versatile and intuitive equivalence-based operational semantics.
## Contents

### Table of Contents

- List of Tables ........................................ vii
- List of Figures ........................................ x
- List of Symbols ....................................... xi

### I Introduction

1. Context ........................................ 1
2. Goals ........................................... 2
3. Overview ......................................... 3
4. Bibliographic Notes .............................. 4

### II Background

5. Constraint Handling Rules ....................... 7
   5.1 Syntax ...................................... 7
   5.2 Semantics .................................. 9
   5.3 Relation to Other Formalisms ............. 17
6. Graph Transformation Systems .................. 19
   6.1 Definitions ................................ 19
   6.2 Semantics ................................ 21
7. Program Analysis ................................ 23
   7.1 Overview .................................. 24
   7.2 Confluence ................................ 26
   7.3 Program Equivalence ......................... 28

### IIIA Complete and Terminating Operational Semantics for Constraint Handling Rules

8. Equivalence-based Operational Semantics .... 32
   8.1 State Equivalence ............................ 33
   8.2 State Transition System ...................... 41
9. Constraint Handling Rules with Persistent Constraints .... 44
   9.1 State Equivalence ............................ 45
   9.2 State Transition System ...................... 47
10. Merge Operator .................................. 50
   10.1 Properties of the Merge Operator ........ 50
10.2 Partial Order on States .................................. 53
11 Discussion ................................................. 54
11.1 Formulations of State Transition System .................. 54
11.2 Range-Restrictedness .................................. 55
11.3 Termination Behavior .................................. 59
11.4 Expressivity ........................................... 60
11.5 Implementation ....................................... 69
12 Related and Future Work ................................ 72

IV Improved Program Analysis Methods 75
13 Invariants .................................................. 75
13.1 Definition .............................................. 75
13.2 Advantages and Disadvantages .......................... 76
13.3 Implications for Program Analysis ....................... 78
14 Confluence .................................................. 81
14.1 Preliminaries ........................................... 81
14.2 Invariant-based Confluence ............................. 84
15 Program Equivalence ..................................... 87
15.1 Preliminaries ........................................... 89
15.2 Reduced Restrictions for Program Equivalence ......... 91
15.3 Invariant-based Program Equivalence ................... 93
16 Related and Future Work ................................ 98

V Embedding Graph Transformation Systems in Constraint Handling Rules 99
17 Encoding of Graph Transformation Systems ............... 99
17.1 Encoding Graphs ....................................... 100
17.2 Encoding Rules ......................................... 101
18 Properties of Encoding ................................... 106
18.1 Graph States ........................................... 106
18.2 Partial Graphs ......................................... 109
18.3 Soundness and Completeness ........................... 109
19 Confluence Analysis ...................................... 113
19.1 Correspondence of Critical Pairs ......................... 114
19.2 Deciding Confluence via Embedding .................... 115
19.3 Discussion ............................................. 120
20 Program Equivalence Analysis ............................ 123
20.1 Redundant Rule Removal ............................... 124
21 Related and Future Work ................................ 125

VI Conclusion ................................................. 127
22 Equivalence-based operational semantics .................. 127
23 Program Analysis ......................................... 128
24 Encoding of Graph Transformation Systems ............... 128

Bibliography .................................................... 131

Index ............................................................ 141
List of Tables

II.1 State Transition System $\omega_{sa}$ ......................................................... 10
II.2 State Transition System $\omega_t$ ................................................................. 12
II.3 State Transition System $\omega_p$ ................................................................. 14

III.1 Comparison of Different State Equivalence Definitions ......................... 35
III.2 State Transition System $\omega_e$ ................................................................. 42
III.3 State Transition System $\omega_e$ with Equivalence Classes ....................... 44
III.4 State Transition System $\omega_!$ ................................................................. 48
III.5 Different Formulations of the Operational Semantics $\omega_e$ over $\Sigma_e$ .... 56
III.6 Different Formulations of the Operational Semantics $\omega_e$ over $\Sigma_e/\equiv_e$ .... 56
III.7 Different Formulations of the Operational Semantics $\omega_!$ over $\Sigma_!$ ....... 57
III.8 Different Formulations of the Operational Semantics $\omega_!$ over $\Sigma_!/\equiv_!$ ...... 58
List of Figures

II.1 Example of a type graph and typed graph .......................... 20
II.2 Double-pushout approach ............................................. 21
II.3 Graph transformation system for recognizing cyclic lists .......... 22
II.4 Simple type graph consisting of a node and edge .................. 22
II.5 Confluence Property .................................................... 26

III.1 Acceptable encodings between different operational semantics .... 62

IV.1 Invariant-based analysis using minimal elements $\sigma_1, \ldots, \sigma_k$ ........ 79

V.1 Cyclic graph consisting of two nodes ................................. 101
V.2 Graph with a dangling edge if node 2 is removed by the \textit{twoloop} rule .... 102
V.3 Example computation ..................................................... 105
V.4 Screenshot of platform for analyzing GTS ............................ 105
V.5 Graph production rule for removing a loop .......................... 115
V.6 A GTS with one-sided track morphism on its critical GTS pair ......... 120
V.7 Graph transformation system – the second rule is redundant ......... 124
V.8 Two operationally non-equivalent graph transformation systems ........ 125
V.9 Derivations for two initial graphs ..................................... 125
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_b$</td>
<td>Conjunction of built-in constraints in rule body</td>
<td>8</td>
</tr>
<tr>
<td>$B_c$</td>
<td>Multiset of CHR constraints in rule body</td>
<td>8</td>
</tr>
<tr>
<td>$G$</td>
<td>(Typed) graph or CHR guard</td>
<td>20</td>
</tr>
<tr>
<td>$H$</td>
<td>(Typed) graph (usually after GTS rule application) or head of CHR rule</td>
<td>21</td>
</tr>
<tr>
<td>$H_1$</td>
<td>Kept head of a CHR rule</td>
<td>8</td>
</tr>
<tr>
<td>$H_2$</td>
<td>Removed head of a CHR rule</td>
<td>8</td>
</tr>
<tr>
<td>$K$</td>
<td>Context graph of a GTS production rule</td>
<td>20</td>
</tr>
<tr>
<td>$L$</td>
<td>Left-hand graph of a GTS production rule</td>
<td>20</td>
</tr>
<tr>
<td>$m$</td>
<td>Match morphism</td>
<td>21</td>
</tr>
<tr>
<td>$R$</td>
<td>Right-hand graph of a GTS production rule</td>
<td>20</td>
</tr>
<tr>
<td>$A$</td>
<td>Set of answers</td>
<td>61</td>
</tr>
<tr>
<td>$B$</td>
<td>Invariant for Blocks World</td>
<td>85</td>
</tr>
<tr>
<td>$C$</td>
<td>Set of constraint symbols</td>
<td>8</td>
</tr>
<tr>
<td>$CT$</td>
<td>CHR constraint theory</td>
<td>8</td>
</tr>
<tr>
<td>$G$</td>
<td>Graph invariant</td>
<td>106</td>
</tr>
<tr>
<td>$I$</td>
<td>Generic invariant</td>
<td>75</td>
</tr>
<tr>
<td>$J$</td>
<td>Invariant for number generator</td>
<td>76</td>
</tr>
<tr>
<td>$M_I$</td>
<td>Set of minimal elements</td>
<td>80</td>
</tr>
<tr>
<td>$NF$</td>
<td>Normal forms</td>
<td>93</td>
</tr>
<tr>
<td>$P$</td>
<td>CHR Program</td>
<td>8</td>
</tr>
<tr>
<td>$R$</td>
<td>Set of graph production rules</td>
<td>20</td>
</tr>
<tr>
<td>$S$</td>
<td>Graph transformation system</td>
<td>20</td>
</tr>
<tr>
<td>$TG$</td>
<td>Type graph</td>
<td>19</td>
</tr>
<tr>
<td>$B$</td>
<td>Conjunction of built-in constraints</td>
<td>12</td>
</tr>
<tr>
<td>$G$</td>
<td>Multiset of CHR constraints</td>
<td>11</td>
</tr>
<tr>
<td>$I$</td>
<td>Interface of CHR programs</td>
<td>92</td>
</tr>
<tr>
<td>$L$</td>
<td>Multiset of linear CHR constraints</td>
<td>46</td>
</tr>
<tr>
<td>$N$</td>
<td>Natural numbers with zero</td>
<td>9</td>
</tr>
<tr>
<td>$P$</td>
<td>Multiset of persistent CHR constraints</td>
<td>46</td>
</tr>
<tr>
<td>$S$</td>
<td>Set of identified CHR constraints</td>
<td>12</td>
</tr>
<tr>
<td>$T$</td>
<td>Propagation history</td>
<td>12</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of global variables</td>
<td>12</td>
</tr>
<tr>
<td>$\omega_{va}$</td>
<td>Very abstract operational semantics</td>
<td>10</td>
</tr>
<tr>
<td>$\omega_c$</td>
<td>Equivalence-based operational semantics</td>
<td>41</td>
</tr>
</tbody>
</table>
Persistent operational semantics ....................................... 47
Theoretical (or abstract) operational semantics .........................12
Refined operational semantics ...........................................13
Operational semantics with rule priorities ..............................14
Set-based operational semantics .........................................15
Set of all states (w.r.t. to arbitrary operational semantics) .......... 34
Set of all \( \omega_{va} \) states ..........................................10
Set of all \( \omega_e \) states ..........................................34
Set of all \( \omega_t \) states ..........................................46
Set of all \( \omega_p \) states ..........................................12
Set of all \( \omega_p \) states ..........................................14
CHR state ..............................................................9
Empty CHR state ..................................................... 34
CHR state (usually after rule application) ...............................9
Generic equivalence relation over CHR states ......................... 34
Equivalence relation over \( \omega_e \) states .............................. 41
Equivalence relation over \( \omega_t \) states .............................. 47
Arbitrary state transition ..............................................10
State transition under \( \omega_{va} \) ............................................. 10
State transition under \( \omega_e \) ............................................. 41
State transition under \( \omega_t \) ............................................. 47
State transition under \( \omega_p \) ............................................. 14
GTS derivation ........................................................21
CHR simplification or simpagation rule ................................. 8
CHR propagation rule .................................................. 8
Graph isomorphism ....................................................20
CHR encoding of a GTS rule ......................................... 101
Joinability of states ...................................................81
CHR merge operator ...................................................50
Partial order on CHR states ..........................................53
Set intersection .........................................................9
Set union ..............................................................9
Subset relation (no equality possible) ....................................9
Subset relation (with possible equality) ..................................9
Multiset union ..........................................................9
Substitution ........................................................... 7
Logical true ............................................................8
Logical false ............................................................8
Syntactic equality .....................................................8
Sequence/List concatenation operator ...................................9
Definition of a symbol ............................................... 9
Chapter I

Introduction

The lurking suspicion that something could be simplified is the world’s richest source of rewarding challenges.

— Edsger Wybe Dijkstra (1930–2002), Computer Scientist

We first present the context for this thesis in Section 1, before we define its goals in Section 2. In Section 3, we give an overview of the structure of this thesis and helpful pointers to dependencies between chapters. Most of this work’s content already appeared in peer-reviewed publications, which are listed in Section 4.

1 Context

Throughout the past years, we witnessed a renaissance of rule- and logic-based approaches. Rule-based programming has been available for decades, with term rewriting (cf. Baader and Nipkow [1998], Ohlebusch [2002] for a summary) being its most prominent representative. Apart from providing a foundation for functional programming, however, direct rule-based programming received little attention outside of research.

In the early 2000s, this situation has changed considerably: we observed a shift from the usage of workflows towards business process management (BPM) (cf. [van der Aalst et al., 2003] for a survey of this change). BPM lacks a rigid formal basis, but could informally be defined as “a generic software system that is driven by explicit process designs to enact and manage operational business processes” [van der Aalst et al., 2003]. A core task involved in BPM is the modeling of business rules that describe process executions. This in turn lead to increased adoption of rule engines and the creation of business rule management systems. A large body of software for these purposes has been made available, including for example, ILOG JRules [ILOG], or JBoss Drools [Drools].

Similarly, logic-based programming approaches are seeing increased usage by companies. Although Prolog, the most prominent logic programming language, has been around since the 70ies (cf. [Sterling and Shapiro, 1994] for an introduction to Prolog), it only had minor impact on the industrial field. Again, we observe a renaissance, which lately culminated in a series of workshops and conferences ([CULP09, CULP10, RulesFest]) specifically targeting industrial practitioners as well as researchers.

Constraint Handling Rules (CHR) is a general-purpose programming language, formally defined as a state transition system, that is both rule- and logic-based. It was initially conceived
2. Goals

by Frühwirth in the early 90ies and continued to evolve during the last two decades [Frühwirth, 1998, Frühwirth and Abdennadher, 2003, Frühwirth, 2009, Sneyers et al., 2010b]. CHR displays the same tendency as many rule- or logic-based approaches in that there already exist hundreds of scientific publications on it, yet we are only just beginning to witness its adoption by industry users.

From the large body of tools available for rule- and logic-based programming, this work focuses on CHR for a number of reasons: Firstly, CHR combines both paradigms in that it is rule-based as well as offering a first-order [Frühwirth, 2009] and linear-logic semantics [Betz and Frühwirth, 2005]. Secondly, CHR is extraordinary among declarative languages, because it allows us to implement any algorithm in optimal time and space complexity [Sneyers et al., 2005].

Furthermore, CHR has been proposed as a lingua-franca for rule-based formalisms, i.e. “a hub which collects and dispenses research efforts from and to the various related fields“ [Sneyers, 2008a]. There already exists a large body of work on comparing CHR with other rule-based approaches (cf. [Frühwirth, 2009, Sneyers et al., 2010b] and Section 5.3), which lead to a cross-fertilization of the corresponding research. Notable results of this include the works by De Koninck [2009] and Van Weert [2009].

The first work by De Koninck [2009] presents an implementation of logical algorithms (LA) in CHR. LA was conceived as a hypothetical language that allows reasoning over runtime behavior of logic programs, and up to now, its only implementation is available through CHR. The work by Van Weert [2009] compares the research results for optimizations available in CHR with other techniques used in the BPM field and finds that CHR consistently outperforms state-of-the-art production rule systems.

2 Goals

For this work, we planned to continue the lingua-franca argumentation by comparing CHR to graph transformation systems (GTSs) [Ehrig and König, 2004]. GTSs are a rule-based formalism that is at the foundation of graph rewriting, as it is applied, for example, for model transformations in the context of model driven development. We selected GTSs in particular due to possible cross-fertilizations with respect to program analysis methods: Firstly, confluence analysis is important for both CHR and GTS, however, it is only decidable for terminating CHR programs [Abdennadher et al., 1996, Plump, 2005]. Secondly, CHR offers a strong result for deciding program equivalence, a topic that has not yet been investigated in the GTS context.

As it befits a potential lingua-franca, we were able to embed GTSs in CHR in a sound and complete fashion [Raiser, 2007]. However, in the process we found this argument to not be as strong as it could. Comparing any formalism with CHR usually involves an embedding with corresponding soundness or completeness results. We have found that the existing definitions of the CHR operational semantics unnecessarily complicate proofs due to the following problems.

**Complex States:** Depending on the operational semantics applied, a CHR state may contain additional components that complicate reasoning over states. For example, the refined operational semantics by Duck et al. [2004] requires identified constraints and tokens, which have no declarative counterparts. On the other hand, the most abstract
formulation, in which a state is simply a conjunction, lacks a means to distinguish global and local variables.

**State Equivalence:** In the GTS context, graph isomorphisms play a central role, however, a corresponding notion for CHR states is not clear. Further research into this matter has revealed (cf. Section 8.1) that there exist various equivalence definitions in the CHR literature, but the implied equivalence relations are incompatible and generally insufficient.

**Operational Semantics:** The theoretical operational semantics of CHR given by Abdenadher [1997] defines the CHR state transition system via four inference rules. However, only two of them correspond to an actual rule application, whilst the remaining ones only modify the states without involving any rule of the program at all. The same problem exists in other formulations, which adds unnecessarily to proof complexity, because there is no clear one-to-one correspondence between rule applications in CHR and another formalism.

Therefore, a first goal of this thesis is to significantly increase the potential for CHR as a lingua-franca by providing a formulation of its operational semantics without these deficiencies. We aim for an elegant formulation of the operational semantics of CHR, however, elegance is in the eye of the beholder. Similarly, there is no clear way to compare different formulations of the CHR operational semantics with respect to their suitability in comparing CHR with other formalisms. Hence, in order to evaluate our proposed formulation, we revisit our initial GTS encoding and investigate the resulting differences.

Furthermore, we plan to extend our initial encoding of GTSs in CHR by considering program analysis in both systems. In this context, the recent introduction of observable confluence by Duck et al. [2007] provides a promising approach towards deciding confluence of real-world programs. We want to revisit observable confluence from the point of view of our proposed operational semantics and investigate its core idea for program equivalence analysis.

Finally, our goal is to apply these extended program analysis methods to our GTS encoding. Hence, we come full circle, by demonstrating the cross-fertilization possibilities implied by the lingua-franca argument.

### 3 Overview

This thesis’ structure consists of three main chapters (III, IV, V). The required knowledge to understand these chapters is given in Chapter II, which is in turn structured corresponding to the main chapters. As this work focuses on Constraint Handling Rules it is inescapable to read Section 5 and Chapter III.

Chapter III introduces our proposed formulation of the operational semantics. In particular, Section 8 serves as the foundation for the remainder of the work. A reader mainly interested in one of the other chapters should at least read this section. The remaining sections of Chapter III may be skipped, except for Section 10, which is required for the later program analysis discussions.

Chapter IV discusses our extensions of existing program analysis methods for CHR. Section 13 generally investigates the role of invariants for program analysis, which are then applied to confluence and program equivalence in Section 14 and Section 15, respectively. A reader
4. Bibliographic Notes

not interested in program analysis may skip this chapter and the corresponding sections in Chapter V completely. Chapter V finally revisits the embedding of graph transformation systems in CHR. Section 17 describes the required encoding, which is then analyzed in Section 18. The sections 19 and 20 investigate CHR program analysis methods in the context of a GTS encoding, which brings together all previously discussed elements. Hence, in order to understand the content of these sections a reader will have to refer to all parts of this thesis.

Each of the main chapters closes with a section that discusses related work and presents potential further lines of research. A final summary of the contents of this thesis are given in Chapter VI.

Additionally, this work sometimes includes detailed discussions of existing works in the literature, or peculiarities of the content that are not relevant for the core topics of the thesis. These parts are correspondingly labeled as excursions.

4 Bibliographic Notes

Most contents of this work already appeared in peer-reviewed publications. Here, we list these publications and give pointers to the corresponding sections of this thesis.

The first encoding of graph transformation systems in CHR is available in [Raiser, 2007]. As explained before, the proof complexity was high due to the formulations of the operational semantics of CHR available at that time. We perceive our simplified formulation of the operational semantics as more elegant, and the changes caused Chapter V to bare little resemblance to its origins in [Raiser, 2007].

We later extended the embedding by a comparison of confluence analysis in GTS and CHR, which appeared in [Raiser and Frühwirth, 2009a,c]. Similarly, we applied a program analysis result of CHR to our encoding in [Raiser and Frühwirth, 2009b].

In parallel, we began our work on simplifying the operational semantics. A first result appeared in [Raiser et al., 2009] and covers an axiomatic state equivalence definition for CHR. It provides the basis for a novel view on the operational semantics as a rewriting system of equivalence classes. We refer to this as the equivalence-based operational semantics $\omega_e$ and discuss it in detail in Section 8.

An implementation for deciding state equivalence based on our results (cf. Theorem 3) is given in [Langbein et al., 2010], where it is used as the foundation of a confluence checker. All previous investigations with regard to graph transformation systems have been reevaluated using the equivalence-based operational semantics, which lead to more elegant and concise definitions and proofs. This combination of all previous results forms the basis of the present thesis and is published in the article [Raiser and Frühwirth, 2010].

Analogously to the abstract operational semantics, $\omega_e$ has no terminating execution model for propagation rules. As our GTS-related research requires no propagation rules, we base the later chapters of this work on $\omega_e$. Nevertheless, we present a possible solution to this problem based on so-called persistent constraints. The work on persistent constraints initially appeared in [Betz et al., 2009]. A refined and extended version first appeared as a technical report in [Betz et al., 2010a] and later as a peer-reviewed article in [Betz et al., 2010b]. It is the basis for Section 9.

Furthermore, this thesis contains material that has not yet been published elsewhere. In particular, this constitutes the definition and investigation of the merge operator in Section 10.
This operator simplifies our discussions of program analysis methods in Chapter IV.

The results on confluence analysis in Section 14 have initially been given by Duck et al. [2007]. However, we found some peculiarities with that work, and hence, reproduced these results by applying our equivalence-based operational semantics and merge operator. We analogously reproduced the work by Abdennadher and Frühwirth [1999] in Section 15, but also extended it in multiple ways.
4. Bibliographic Notes
Chapter II

Background

In this chapter, we provide the reader with the required background knowledge about Constraint Handling Rules (Section 5), graph transformation systems (Section 6), and program analysis (Section 7).

5 Constraint Handling Rules

Constraint Handling Rules has been introduced by Frühwirth in the early 90ies. The article [Frühwirth, 1998] summarized its early development and was the main reference for the following decade. It was recently replaced by a new book from Frühwirth [2009], which also serves as the foundation of this section.

5.1 Syntax

We first describe general syntactic prerequisites in Section 5.1.1, before we detail CHR-specific syntax in Section 5.1.2. Finally, Section 5.1.3 presents the notations and abbreviations used throughout this work.

5.1.1 General

The syntax of CHR is based on a set of variables \( V \), a set of function symbols \( \Phi \), and a set of predicate symbols \( \Pi \). Symbols from the latter two sets are associated with an *arity* for the number of arguments they take and we write \( c/n \) for a function or predicate symbol \( c \) with arity \( n \). Functions with arity 0 are called *constants*. Functions with arity 1 are called *unary functions*.

We recursively define a *term* as being either a variable or a *function term* of the form \( f(t_1, \ldots, t_n) \) where \( f/n \in \Phi \) and all \( t_i(1 \leq i \leq n) \) are terms. A term, in which no variables occur, is called *ground*. A *substitution* \( \Theta \) is a finite function from variables to terms written as \( \Theta = \{ X_1/t_1, \ldots, X_n/t_n \} \) where each \( X_i \neq t_i \). For an expression \( E \) and substitution \( \Theta \), the expression \( E\Theta \) is given by simultaneously substituting each occurrence of a variable \( X_i \) in \( E \) by the term \( t_i \) for each \( X_i/t_i \in \Theta \). \( E\Theta \) is called an *instance* of \( E \). Two expressions \( E \) and \( F \) are *variants* of each other if one is an instance of the other. Variants can be obtained by a bijective renaming of variables.
5. Constraint Handling Rules

5.1.2 Syntax of Constraint Handling Rules

For CHR, the set \( \Pi \) of predicate symbols is disjointly made from a set of \textit{CHR constraint symbols} and a set of \textit{built-in constraint symbols}. For \( c \in \Pi \) and terms \( t_i (1 \leq i \leq n) \) we call \( c(t_1, \ldots , t_n) \) a \textit{CHR constraint} if \( c \) is a \textit{CHR constraint symbol}, and a \textit{built-in constraint} (or just \textit{built-in}), otherwise. The set of all CHR constraint symbols occurring in all rules of a CHR program \( \mathcal{P} \) is denoted as \( C(\mathcal{P}) \).

The set of built-in constraint symbols contains three distinguished symbols: \( \top \) (true), \( \bot \) (false), and = (syntactic equality). CHR requires a logical theory \( CT \), called \textit{constraint theory}, to reason over built-ins. A constraint theory can be freely selected by developers, as long as it is consistent, complete, and includes an axiomatization for the above distinguished symbols. This axiomatization has to satisfy that \( \top \) always holds, \( \bot \) never holds, and = holds for syntactically equivalent terms. Practically, the constraint theory is realized by the host language underlying CHR.

For any term, predicate, CHR constraint, built-in constraint, and set of variables, \( \text{vars}(\cdot) \) denotes its free variables. There exist different definitions of CHR states, however, in each case a set of \textit{global} variables of a CHR state is defined. All variables occurring in a CHR state that are not global are \textit{local} variables and those local variables, which are only used for built-in constraints are referred to as \textit{strictly local} variables.

\textbf{Definition 5.1 (CHR Rules).} A CHR program \( \mathcal{P} \) is a finite set of CHR rules (or just rules if the context is unambiguous) of the following form

\[ H_1 \setminus H_2 \leftrightarrow G \mid B, \]

where

- \( H_1 \) and \( H_2 \) – called the head – are multisets of CHR constraints,
- \( G \) – called the guard – is a conjunction of built-ins, and
- \( B = B_c, B_b \) – called the body – consists of a multiset \( B_c \) of CHR constraints and a conjunction \( B_b \) of built-ins.

\( H_1 \) is referred to as the kept head and \( H_2 \) as the removed head. There are different specializations of CHR rules, depending on the head:

- Simpagation rules \( (H_1 \neq \emptyset \land H_2 \neq \emptyset) \) written as \( H_1 \setminus H_2 \leftrightarrow G \mid B \)
- Simplification rules \( (H_1 = \emptyset \land H_2 \neq \emptyset) \) written as \( H_2 \leftrightarrow G \mid B \)
- Propagation rules \( (H_1 \neq \emptyset \land H_2 = \emptyset) \) written as \( H_1 \Rightarrow G \mid B \)

Every CHR rule can optionally be preceded by an identifier (or name) followed by @. Finally, if the guard \( G \) is \( \top \) it may be omitted together with the \( \mid \) character.

Variables that occur in the rule, but not in its head, are called local variables. A rule without local variables is called range-restricted and a CHR program is called range-restricted if all its rules are range-restricted.
5.1.3 Notations and Abbreviations

We take the following liberties for writing CHR rules: Firstly, for sets and multisets we omit the curly brackets and instead write all elements separated by commas if no ambiguity can occur. Secondly, we freely mix built-ins and CHR constraints in bodies and separate them by commas. Thirdly, for an empty conjunction of built-ins in the body we write $\top$, except for bodies containing CHR constraints, in which case we simply omit the built-ins.

Example 5.2 (Exemplary CHR rules). The following are possible CHR rules:

\[
\begin{align*}
gcd(0) & \iff \top \\
gcd(N) \setminus \gcd(M) & \iff N > 0 \land M \geq N | L = M \% N, \gcd(L) \\
transitive & @ \text{edge}(A, B), \text{edge}(B, C) \implies \text{edge}(A, C)
\end{align*}
\]

Furthermore, this work makes use of the following notations:

- $\uplus$ (infix) operator for multiset union
- $\cup$ (infix) operator for set union
- $\cap$ (infix) operator for set intersection
- $\subseteq$ the subset relation with possible equivalence and $\subset$ for actual subsets.
- $[H \mid T]$ for a sequence/list with head $H$ and remaining sequence/list $T$
- $[]$ for the empty sequence/list
- $++$ (infix) operator for sequence/list concatenation
- $N$ as usual denotes the natural numbers including zero.
- We use $\sigma, \sigma', \sigma_1, \sigma_2, \ldots$ and $\tau, \tau', \tau_1, \tau_2, \ldots$ for CHR states, where the $\tau$-variants are usually states resulting from a rule application.
- We use $S ::= e$ for defining a new symbol $S$ as the expression $e$.

5.2 Semantics

We mentioned earlier that CHR has both, declarative and operational, semantics. There are two declarative semantics for CHR, which we present in Section 5.2.5: the original declarative semantics based on first-order logic [Frühwirth, 1998] and a more recent formulation using linear-logic [Betz and Frühwirth, 2005].

The operational semantics of CHR is closer to the implementational level and has received more attention in the past. Therefore, there are now several different formulations and extensions of it available. In Sections 5.2.1, 5.2.2, and 5.2.3, we discuss the very abstract, theoretical, and rule-priority operational semantics in detail, which are used throughout this work. Section 5.2.4 provides an overview of further operational semantics.

All operational semantics are defined as state transition systems, i.e. a binary relation over states. The following definition provides shared nomenclature for further treatment of state transition systems.
5. Constraint Handling Rules

Apply:
\[ r @ H_1 \setminus H_2 \leftrightarrow G \mid B \text{ an instance of a rule in } \mathcal{P} \text{ with new variables } \bar{x} \]
\[ CT \models \forall (G \rightarrow \exists \bar{x}. G) \]
\[ (H_1 \land H_2 \land G) \Rightarrow_{va} (H_1 \land G \land B \land G) \]

Table II.1: State Transition System \( \omega_{va} \)

Definition 5.3 (State Transition System [Frühwirth, 2009]). A state transition system \( T \) is a pair \((S, \rightarrow)\), where \( S \) is a set of states (configurations) and \( \rightarrow \subseteq S \times S \) is a binary relation on states, called the transition relation.

A transition system is deterministic if there is at most one transition from every state, i.e. the transition relation is a partial function; otherwise it is non-deterministic.

The reachability relation \( \rightarrow^* \) is the reflexive-transitive closure of \( \rightarrow \) and \( \rightarrow^k \) for a number \( k \) is defined recursively as \( \rightarrow^1 \equiv \rightarrow \) and \( \rightarrow^{k+1} \equiv \rightarrow \circ \rightarrow^k \).

Initial states and final states are non-empty subsets of the set of states. Every state from which no transition is possible is final.

A derivation (or computation) is a sequence of states \( \sigma_0, \sigma_1, \ldots \), often written \( \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \), such that \( \sigma_0 \rightarrow \sigma_1 \land \sigma_1 \rightarrow \sigma_2 \land \ldots \). A derivation is finite (terminating) if its sequence is finite, otherwise it is infinite (diverging, non-terminating). The derivation length is the number of transitions in a derivation.

Depending on context, a transition is also called a derivation step or computation step.

5.2.1 Very Abstract Operational Semantics

The very abstract operational semantics of CHR, denoted as \( \omega_{va} \), was first formulated by Frühwirth [2009]. Its goal is to capture the essence of CHR’s multiset rewriting, whilst remaining as generic as possible. All other operational semantics can be considered as specializations of \( \omega_{va} \). This generality, however, comes at a cost: \( \omega_{va} \) lacks a terminating execution model as well as execution control mechanisms. We will discuss the impact of these problems, after we have fully defined \( \omega_{va} \). The following definitions are adapted from Frühwirth [2009].

Definition 5.4 (\( \omega_{va} \)-State). A \( \omega_{va} \)-state is a conjunction of built-in and CHR constraints. An initial \( \omega_{va} \)-state is an arbitrary \( \omega_{va} \)-state. A final \( \omega_{va} \)-state is one where no \( \omega_{va} \)-transition is possible anymore. The set of all \( \omega_{va} \)-states is denoted as \( \Sigma_{va} \).

The formulation of the operational semantics \( \omega_{va} \) assumes all rules to be in head normal form in which all arguments of constraints in the head are unique variables. Every rule can be transformed into head normal form by replacing arguments \( t_i \) with new variables \( V_i \) and adding \( V_i = t_i \) to the guard of the rule.

Definition 5.5 (\( \omega_{va} \)-Transitions). The state transition system \((\Sigma_{va}, \Rightarrow_{va})\), referred to as \( \omega_{va} \), is given in Table II.1.

Example 5.6. Consider the following CHR program for computing the greatest common divisor. It assumes that the constraint theory CT handles inequalities (in particular > and
and modular arithmetic (%) in addition to syntactic equality.

\[
\begin{align*}
gcd1 \equiv & \ gcd(N) \land N = 0 \\
gcd2 \equiv & \ gcd(N) \land gdc(M) \land N > 0 \land M \geq N \land L = M \% N, gcd(L)
\end{align*}
\]

The following transitions demonstrate the computation of the greatest common divisor of the numbers 6 and 9, based on the example given by Frühwirth [2009, p.57]:

\[
\begin{align*}
\langle gcd(6) \land gcd(9) \rangle & \rightarrow gcd2 \langle gcd(6) \land 6 > 0 \land 9 \geq 6 \land L = 9 \%% 6 \land gcd(L) \rangle \\
\langle gcd(L) \land L > 0 \land 6 \geq L \land L' = 6 \%% L \land gcd(L') \land 6 > 0 \land 9 \geq 6 \land L = 9 \%% 6 \rangle & \rightarrow gcd2 \langle \land G \rangle \\
\langle \land L' = 0 \land gcd(L) \land L > 0 \land 6 \geq L \land L' = 6 \%% L \land 6 > 0 \land 9 \geq 6 \land L = 9 \%% 6 \rangle & \rightarrow gcd1 \langle L' = 0 \land gcd(L) \land L > 0 \land 6 \geq L \land L' = 6 \%% L \land 6 > 0 \land 9 \geq 6 \land L = 9 \%% 6 \rangle
\end{align*}
\]

The above transitions differ significantly from the example given by Frühwirth [2009]. This is due to an implicit simplification made by Frühwirth [2009] that improves readability. However, strict application of Definition 5.5 leads to the above, less readable, transitions. Nevertheless, the result remains the same: the greatest common divisor of 6 and 9 is 3 (slightly obscured above as \(gcd(L) \land L = 9 \%% 6\)).

The very abstract operational semantics provides no execution control mechanism. Selecting an applicable rule is non-deterministic, as well as the selection of the involved CHR constraints. Additionally, \(\rightarrow_{va}\) suffers from the so-called trivial non-termination problem: for a \(\omega_{va}\)-state \(\sigma\) and a propagation rule \(r\) with \(\sigma \rightarrow_{va} \sigma_1\) the \(\omega_{va}\)-state \(\sigma_1\) and rule \(r\) satisfy the conditions for Apply, hence, \(\sigma_1 \rightarrow_{va} \sigma_2\). Again, rule \(r\) can be applied to \(\sigma_2\) such that \(\sigma_2 \rightarrow_{va} \sigma_3\) and so on, ad infinitum. Therefore, any propagation rule in a CHR program potentially leads to an infinite sequence of transitions. Similarly, inconsistent built-ins imply any formula, hence, the condition in Apply can always be satisfied, leading to the application of further rules and hence possibly to non-termination.

5.2.2 Theoretical Operational Semantics

The theoretical operational semantics \(\omega_t\) eliminates the trivial non-termination problem found in \(\omega_{va}\). It’s core idea is that a propagation rule can only be fired once for a specific combination of CHR constraints. This is realized by identifying CHR constraints using integer numbers and keeping a store of (propagation) tokens that represents previously applied rules.

The CHR constraint identifiers are required because of the multiset semantics of CHR: For two CHR constraints with the same CHR constraint symbol and arguments, an applicable propagation rule needs to be applicable to each of them.

**Definition 5.7 (Identified CHR Constraint).** An identified CHR constraint \(c\#i\) is a CHR constraint \(c\) associated with some unique identifier \(i\), the constraint identifier. We introduce the functions \(\text{chr}(c\#i) = c\) and \(\text{id}(c\#i) = i\), and extend them to sequences and sets of identified CHR constraints in the obvious manner.

Due to the need to identify CHR constraints, \(\omega_t\) relies on a different state definition, which separates the goal from identified CHR constraints.

**Definition 5.8 (\(\omega_t\)-States).** A \(\omega_t\)-state \(\sigma\) is a tuple of the form \((G; S; \mathbb{B}; T)^V_N\) where

- the goal (store) \(G\) is a multiset of (CHR and built-in) constraints,
5. Constraint Handling Rules

Solve:
\[ c \text{ is a built-in constraint} \]
\[ C T \models \forall ((c \land B) \leftrightarrow B') \]
\[ \langle \{ c \} \cup G; S; B; T \rangle_n \text{ } \rightarrow_r^T \text{ solve } \langle G; S; B'; T \rangle_n \]

Introduce:
\[ c \text{ is a CHR constraint} \]
\[ \langle \{ c \} \cup G; S; B; T \rangle_n \text{ } \rightarrow_r^T \text{ introduce } \langle G; \{ c \# n \} \cup S; B; T \rangle_{n+1} \]

Apply:
\[ r \# H_1 \setminus H_2 \leftrightarrow G \mid B \text{ a variant of a rule in } \mathcal{P} \text{ with fresh variables } \bar{x} \]
\[ C T \models \exists (B) \land \forall (B \rightarrow \exists \bar{x}. (\text{chr}(H_1) = H_1' \land \text{chr}(H_2) = H_2' \land G)) \]
\[ (r, \text{id}(H_1) \# \# \text{id}(H_2)) \notin T \]
\[ \langle B \cup G; H_1 \cup S; \text{chr}(H_1) = H_1' \land \text{chr}(H_2) = H_2' \land G \land B; T \cup \{ (r, \text{id}(H_1) \# \# \text{id}(H_2)) \} \rangle_n^V \]

Table II.2: State Transition System \( \omega_t \)

- the CHR (constraint) store \( S \) is a set of identified CHR constraints,
- the built-in (constraint) store \( B \) is a conjunction of built-in constraints,
- the propagation history \( T \) is a set of tuples \( (r, I) \) for a rule name \( r \) and a sequence \( I \) of constraint identifiers,
- the counter \( n \) represents the next free integer usable as constraint identifier, and
- the set of global variables \( V \) contains the variables of the initial goal.

The set of all \( \omega_t \)-states is denoted as \( \Sigma_t \).

The Apply rule of the state transition system for the operational semantics \( \omega_t \) takes the propagation history into account. Furthermore, the rules Solve and Introduce transfer constraints from the goal store into the CHR or built-in stores, respectively. The definition given below is based on the formulation by Frühwirth [2009].

**Definition 5.9 (\( \omega_t \)-Transitions).** The state transition system \((\Sigma_t, \rightarrow_t)\), referred to as \( \omega_t \), is given in Table II.2. We simply write \( \sigma \rightarrow_t \tau \) if the involved inference rule is of no importance.

For Definition 5.9, we require that the program \( \mathcal{P} \) contains no rules with the names Solve or Introduce, such that Apply transitions can be clearly identified. Otherwise, it is possible to annotate Apply transitions with “apply” and the involved rule name (as in [Frühwirth, 2009]), which we avoided for brevity.

**Example 5.10.** Reconsider the CHR program for computing greatest common divisors from Example 5.6. The following derivation demonstrates the program’s execution under the oper-
ational semantics \( \omega_1 \) for computing the greatest common divisor of 6 and 9.

\[
\begin{align*}
\rightarrow & \text{Introduce} \quad \langle \gcd(6), \gcd(9); \top; \top \rangle_0^\emptyset \\
\rightarrow & \text{Introduce} \quad \langle \gcd(9); \gcd(6) \# 0; \top; \top \rangle_1^\emptyset \\
\rightarrow & \gcd_2 \quad \langle 0; \gcd(6) \# 0, \gcd(9) \# 1; \top; \top \rangle_2^\emptyset \\
\rightarrow & \text{Solve} \quad \langle \gcd(L); \gcd(6) \# 0; L = 3; (\gcd_2, [0, 1]) \rangle_2^\emptyset \\
\rightarrow & \text{Introduce} \quad \langle 0; \gcd(6) \# 0; \gcd(L) \# 2; L = 3; (\gcd_2, [0, 1]) \rangle_3^\emptyset \\
\rightarrow & \gcd_2 \quad \langle L' = M' \# N', \gcd(L'); \gcd(L) \# 2; L = N' \# 6 = M' \# N' > 0 \wedge M' \geq N' \wedge L = 3; \\
& \quad (\gcd_2, [0, 1]), (\gcd_2, [2, 0]) \rangle_3^\emptyset \\
\rightarrow & \text{Solve} \quad \langle \gcd(L'); \gcd(L) \# 2; L = 3 \wedge L' = 0; (\gcd_2, [0, 1]), (\gcd_2, [2, 0]) \rangle_4^\emptyset \\
\rightarrow & \gcd_1 \quad \langle 0; \gcd(L) \# 2, \gcd(L') \# 3; L = 3 \wedge L' = 0; (\gcd_2, [0, 1]), (\gcd_2, [2, 0]), (\gcd_1, [3]) \rangle_4^\emptyset \\
\end{align*}
\]

Example 5.10 highlights important properties of the operational semantics \( \omega_1 \): Firstly, there are more sources of non-determinism in \( \omega_1 \) than in \( \omega_{\text{rp}} \). Additional non-determinism occurs in the multiset-based selection of a constraint from the goal store, i.e. selection between \textit{Solve} and \textit{Introduce} is non-deterministic. Furthermore, an arbitrary built-in store \( \mathbb{B}' \) is chosen for each \textit{Solve} transition. In the above example \( \mathbb{B}' \) was chosen to optimize readability of the resulting built-in store. However, the final state of the example reveals that this kind of optimization is not allowed at every point in the derivation, but is strictly limited to \textit{Solve} transitions.

Secondly, the propagation history contains numerous irrelevant tokens in this formulation. As simplification and simpagation rules remove a constraint with a unique identifier, the corresponding token will never hinder another rule application. Therefore, in practice, only the application of a propagation rule results in a token creation. Additionally, tokens may remain in the propagation history, although the constraint identifiers may point to an identified CHR constraint that has already been removed.

**Refined Operational Semantics** The theoretical operational semantics is the foundation for the so-called \textit{refined operational semantics} \( \omega_r \) introduced by Duck et al. [2004]. Its purpose is to formally characterize the behavior of CHR implementations, i.e. it mainly reduces sources of non-determinism contained in \( \omega_1 \). In particular, the choice of which rule to apply is made by examining rules for applicability in textual order. Similarly, an active constraint is determined from left to right in the goal.

In the remainder of this work, we may sometimes refer to \( \omega_r \), but we generally prefer to consider \( \omega_1 \). The refined operational semantics is based on the same essential ideas as \( \omega_1 \), but although it is focused on being close to implementations it is still not a characterization of any existing implementation. For example, the refined operational semantics still contains non-determinism when selecting suitable partner constraints for a rule head matching. For these reasons, we refrain from presenting the inference rules for \( \omega_r \) and instead refer the reader to [Duck et al., 2004] for more details.

### 5.2.3 Operational Semantics with Rule Priorities

De Koninck et al. [2007] extended CHR with an additional execution control in the form of rule priorities. The resulting operational semantics is referred to as \( \omega_{\text{rp}} \) and instead of normal CHR programs it is based on CHR\textsuperscript{IP} programs, in which each rule has an additional
5. Constraint Handling Rules

Solve:
\[
\frac{c \text{ is a built-in constraint}}{\langle \{c\} \cup G; S; B; T \rangle_n \xrightarrow{\text{Solve}} \langle G; S; B'; T \rangle_n}
\]

Introduce:
\[
\frac{c \text{ is a CHR constraint}}{\langle \{c\} \cup G; S; B; T \rangle_n \xrightarrow{\text{Introduce}} \langle G; \{c\#n\} \cup S; B; T \rangle_{n+1}}
\]

Apply:
\[
\frac{p :: r @ H_1 \setminus H_2 \iff G \mid B \text{ a rule in } P \text{ with new variables } \bar{x}}{\langle \emptyset; H_1 \cup H_2 \cup S; B; T \rangle_n \xrightarrow{\text{Apply}} \langle B; H_1 \cup S; B \land G; T \cup (r, \text{id}(H_1) ++ \text{id}(H_2)) \rangle_n}
\]

Table II.3: State Transition System \(\omega_p\)

priority annotation. Rule priorities allow for more fine-grained execution control, as the non-deterministic choice between applicable rules is replaced by choosing an applicable rule of highest priority.

In this work we use the rule notation from De Koninck et al. [2008], i.e. a CHR\(^{rp}\) rule has the form

\[
p :: r @ H_1 \setminus H_2 \iff G \mid B
\]

where \(p\) is a priority expression. Priorities can be specified either as constants, or as expressions depending on variables present in the head constraints. In the latter case, we refer to the rule as a dynamic priority rule, otherwise it is a static priority rule.

**Definition 5.11 (\(\omega_p\)-States).** A \(\omega_p\)-state is a \(\omega_t\)-state. The set of all \(\omega_p\)-states is denoted as \(\Sigma_p\).

CHR\(^{rp}\) shares its state definition with \(\omega_t\), as well as most of the state transition system. There are two significant differences in the Apply rule: Firstly, the inference rule is extended to take priorities into account and secondly, it is only applicable when the goal store is empty. This restriction is necessary to be able to determine a rule of highest priority, as CHR constraints in the goal might cause a higher priority rule application otherwise.

**Definition 5.12 (\(\omega_p\)-Transitions [De Koninck et al., 2008]).** The state transition system \((\Sigma_p, \xrightarrow{\rho})\), referred to as \(\omega_p\), is given in Table II.3. We simply write \(\sigma \xrightarrow{\rho} \tau\) if the involved inference rule is of no importance.

The additional execution control gained by rule priorities, often allows us to express algorithms in a more concise and readable way. In [Gabrielli et al., 2009] this increase of expressivity has been formally proven.

Definition 5.12 is based on the formulation given by De Koninck et al. [2008] and further shows the variance in different possible formulations for CHR operational semantics. For example, a substitution is used instead of referring to rule instances. Furthermore, the formulation by
De Koninck et al. [2008] allows the usage of Apply for inconsistent built-in stores, which can lead to the problems mentioned above.

Example 5.13. The expressive power of CHR\(^p\) can be seen in the following example program, which implements the Dijkstra algorithm for computing shortest paths.

\[
\begin{align*}
1 & : \text{source}(V) \implies \text{dist}(V,0) \\
1 & : \text{dist}(V,D_1) \setminus \text{dist}(V,D_2) \iff D_1 \leq D_2 | \top \\
D + 2 & : \text{dist}(V,D), \text{edge}(V,C,U) \implies \text{dist}(U,D + C)
\end{align*}
\]

For this program the graph is represented using edge \(\setminus3\) constraints where the arguments denote source, cost, and target, respectively. The algorithm is initialized with a source\((V)\) constraint, that creates the first dist\(\setminus2\) constraint. A dist\((V,D)\) constraint represents that the shortest path from the source to node \(V\) is of length \(D\). The second rule ensures that only the shortest paths are stored in dist\(\setminus2\) constraints. Finally, the third rule extends the available shortest paths. Its dynamic priority orders the distance updates as required by Dijkstra’s algorithm.

5.2.4 Other Operational Semantics

The CHR community has formulated a plethora of operational semantics and extensions for CHR. The semantics discussed above are used throughout the remainder of this work and in this section we provide an (incomplete) overview of other operational semantics and extensions of CHR available in the literature.

Set-based Although CHR is primarily considered as a language for multiset rewriting, there is also an operational semantics \(\omega_{\text{set}}\) available based on set rewriting. It is presented in [Sarna-Starosta and Ramakrishnan, 2007], which changes two other significant aspects of CHR as well: Firstly, tabled execution is considered in analogy to tabled Prolog variants. Secondly, the trivial non-termination problem is solved in a unique way different from the propagation history. The latter change is discussed in more detail in Chapter III.

Disjunctive The committed-choice nature of CHR means that computations are made without backtracking, i.e. a rule application is never reversed or undone. This complicates formulating search algorithms in CHR. An extension which allows disjunctive bodies, and hence, backtracking over these alternatives, is called CHR\(^\lor\) [Abdennadher and Schütz, 1998, Abdennadher, 2001].

Probabilistic Another important extension of CHR allows probabilistic elements. PCHR [Frühwirth et al., 2002] is an early formulation that allows probabilistic selection of applicable rules. This idea has been taken further in the work on CHrisM [Sneyers et al., 2009a, 2010a], which is a combination of CHR with the probabilistic Prolog dialect PRISM.

Concurrent CHR in its abstract form is well-suited for concurrent execution and prototype implementations exist which exploit this property. Sulzmann and Lam [2007] introduced a concurrent implementation in Haskell. It uses Haskell’s support for shared transaction memory to resolve conflicting rule applications. The implementation was further refined in [Sulzmann and Lam, 2008].
CHR\(^2\) A very recent proposal for a new operational semantics, called CHR\(^2\), has been given by Van Weert [2010] in his thesis. It offers a combination of many features of existing extensions of CHR, like aggregates, rule priorities, or batch processing. However, there only exists a prototype implementation of CHR\(^2\) and a glance at the inference rules given in [Van Weert, 2010] immediately reveals that it tremendously complicates formal analysis. As the goal of our thesis is to find an elegant formulation of the operational semantics, we will not discuss CHR\(^2\) any further. However, the results given in Chapter III can be extended analogously to existing CHR extensions, such as to come closer to an elegant formal representation of the features of CHR\(^2\).

More details on available formulations and extensions can be found in [Frühwirth, 2009] and the surveys [Frühwirth, 1998, Sneyers et al., 2010b].

5.2.5 Declarative Semantics

We already stated in the introduction that CHR is a logic-based language. This means, that every CHR rule has a logical reading, as well as every CHR state. In addition, execution of a CHR program corresponds to a proof that the final state is logically implied by the initial state.

The initial definitions of the declarative semantics of CHR are based on first-order logic. In the original context of CHR as a language for implementing constraint solvers, this made perfect sense. However, when using CHR as a general-purpose language the first-order logic is insufficient for two reasons.

**Multiset Semantics:** Operationally, CHR is defined with a multiset semantics such that multiplicity of a constraint matters. Due to the equivalence \(a \land a \leftrightarrow a\) generally holding in first-order logic, we cannot account for multisets in the declarative semantics. Hence, considering a first-order logical reading of any program or state containing multisets is infeasible.

**Non-Monotonicity:** In a general-purpose program, developers can write rules that explicitly violate equivalence of heads and bodies. A typical example is given below, which demonstrates that a first-order logical reading of such programs is meaningless.

To compensate for these problems, Betz and Frühwirth [2005] introduced the linear-logic declarative semantics for CHR. It is based on the concept of consumable resources, and hence, supports both, multiset semantics and non-monotonicity.

**First-Order Logic** Every CHR rule has a logical reading in first-order logic according to the following definition based on [Frühwirth, 2009].

**Definition 5.14** (First-Order Logical Reading of Rules). The logical reading of the three types of rules is as follows, where \(\bar{y}\) are the local variables of each rule.

\[
\begin{align*}
\text{Simplification:} & \quad H \leftrightarrow G \mid B \quad \forall(G \rightarrow (H \leftrightarrow \exists \bar{y}.B)) \\
\text{Propagation:} & \quad H \implies G \mid B \quad \forall(G \rightarrow (H \rightarrow \exists \bar{y}.B)) \\
\text{Simpagation:} & \quad H_1 \setminus H_2 \leftrightarrow G \mid B \quad \forall(C \rightarrow ((H_1 \land H_2) \leftrightarrow (H_1 \land \exists \bar{y}.B)))
\end{align*}
\]
Example 5.15. As stated above, non-monotonicity is problematic with first-order logical reading for rules. To this end, consider the following program that exploits CHR’s non-determinism to realize a coin throw.

\[
\begin{align*}
\text{coin} & \leftrightarrow \top \\
\text{coin} & \leftrightarrow \bot
\end{align*}
\]

Operationally, the meaning of this program is intuitively understandable, however, consider the following logical reading of the program, which is the conjunction of the logical readings of all rules.

\[
\text{coin} \leftrightarrow \top \land \text{coin} \leftrightarrow \bot
\]

Clearly, the program’s logical reading is inconsistent, as we can derive from the above that \(\top \leftrightarrow \bot\).

In addition to rules, CHR states have a logical reading as well, given by the following definition.

**Definition 5.16** (First-Order Logical Reading of States). The logical reading of a \(\omega_1\)-state of the form \(\langle G; S; B; T \rangle\) is the formula

\[
\exists \bar{y}. (G \land \text{chr}(S) \land B),
\]

where \(\bar{y}\) are the local variables of the state.

Previously, we objected to the complexity of CHR states (cf. Section 2), and indeed, the above definition clearly shows that two components of a \(\omega_1\)-state are irrelevant for its logical reading. Furthermore, the distinction between \(G\) and \(S\), namely constraint identifiers, is eliminated again in the logical reading.

For a more detailed discussion of the first-order declarative logic of CHR, including soundness and completeness results, we refer the interested reader to [Frühwirth, 2009].

**Linear-Logic** In order to eliminate the above problems with a first-order declarative reading, [Betz and Frühwirth, 2005] introduced a linear-logic declarative reading. Linear-logic is based on the idea of resources that can be consumed, hence, in the above coin throw example the linear-logic reading remains useful. It captures the idea of a coin resource being consumed and integrates the committed-choice nature of CHR.

We will refrain from discussing the linear-logic semantics in this work, as no knowledge of it is required for the remainder. Instead, we elaborate on the two points relevant to our work. Firstly, the linear-logical reading of a \(\omega_1\)-state displays the same problems with respect to state complexity as its reading in first-order semantics. Secondly, in intuitionistic linear-logic a “bang”-operator is defined, which intuitively makes a resource available any number of times. In particular, a banged resource is not consumed during reasoning. We transfer this idea to CHR, when we introduce persistent constraints in Section 9.

5.3 Relation to Other Formalisms

The proposal of CHR as a lingua franca for rewriting systems arrived after numerous comparisons between CHR and other formalisms have been undertaken. In this section, we recapitulate selected results from this line of research. However, as this thesis focuses on graph transformation systems, for which no prior work existed, we refrain from an extensive discussion and instead provide the interested reader with pointers to the corresponding literature.
Set-based Formalisms

In Section 1, we mentioned business rules, which are a RETE-based paradigm. A comprehensive comparison with CHR, focusing on efficient execution, was given by Van Weert [2009]. Business rules have influenced the early CHR compiler implementations, but the current systems are based on numerous research results for improving efficiency. A summary of the techniques applied in CHR compilation is given by [Schrijvers, 2005] in his PhD thesis. Together with more recent optimizations, Van Weert [2009] has shown that execution of CHR is faster than traditional RETE-based systems by several magnitudes.

We also already mentioned the work by De Koninck [2009], who investigated the close relation between CHR and logical algorithms (LA). LA is a formalism proposed by Ganzinger and McAllester [2002] in order to alleviate the problem of determining runtime complexity of logic programs. De Koninck [2009] successfully embedded LA into CHR with rule priorities, which allowed him to transfer a meta-complexity result, available for LA, to a subset of CHR with rule priorities. Additionally, he provided a mapping from CHR with rule priorities into regular CHR, which in turn made his embedding the first actual implementation of LA.

Logical Formalisms

Apart from the already discussed first-order and linear-logic declarative semantics, other logical formalisms have been considered. This line of research was pursued mainly by Meister [2008]. He implemented a fragment of frame-logic, which is an object-oriented extension of classical first-order logic, in CHR. Furthermore, he gave a transaction logic semantics for CHR.

Term Rewriting

CHR is closely related to term rewriting. The main difference is that CHR rewrites a flat set of constraints, whereas in term rewriting nested terms can be rewritten. It is possible to simulate term rewriting in CHR by a simple flattening function, as explained in [Frühwirth, 2009]. However, this only works for linear term rewriting systems. Raiser and Frühwirth [2008] instead considered term graph rewriting and provided the necessary folding rules in CHR for sharing term structures. This work is based on prior research by Plump [1993] on jungle evaluation, and a general treatment of term graph rewriting is available in [Ohlebusch, 2002].

Term rewriting is the basis of functional programming and CHR has also been applied to the problem of type checking and inference [Alves and Florido, 2002, Stuckey et al., 2006]. Finally, Martinez [2010] compared linear concurrent constraint programming with CHR, which resulted in an encoding of the λ-calculus in CHR.

Graph-based Formalisms

We are especially interested in prior work with respect to graph-based formalisms, as this work deals with graph transformation systems. However, there exist only few research results in this direction.

The comparison with term graph rewriting was already mentioned above. Another interesting work is given by Betz [2007], who compared colored Petri nets to CHR. A subset of
these can be translated into CHR, and there also exists a sound and complete encoding of place/transition nets.

**Other Formalisms**

We explained above that we will not elaborate on all formalisms that have been compared to CHR. For more complete surveys, the reader should consult [Frühwirth, 2009] and [Sneyers et al., 2010b]. In addition to the above, these discuss comparisons of CHR with ACD term rewriting, Join-Calculus, equivalent transformation rules, production rules, event-condition-action rules, GAMMA, functional programming, deductive databases, Prolog, (concurrent) constraint logic programming, and more.

6 Graph Transformation Systems

In this section, we introduce the required definitions for graph transformation systems. In particular, we consider typed GTSs, i.e. each node and edge in a graph is assigned a certain type. To this end, we distinguish a type graph, which holds all possible types and any number of typed graphs, which are graphs that use types from a type graph. The formal definitions for these are given in Section 6.1, before we introduce the category-theoretical semantics in Section 6.2

In our work, we focus on the so-called double-pushout approach, yet we give a concise introduction into its alternative, the single-pushout approach. The following definitions for algebraic graph transformation systems have been adapted from Ehrig et al. [2006]. The semantics of GTSs is defined via pushouts in category theory. Although we present it in this way, we will mostly refer to a set-based interpretation, as most readers will be familiar with that. For an interested reader, [Ehrig et al., 2006] contains an introduction to category theory, and in particular, pushouts.

6.1 Definitions

We begin with the definition of a graph, which is then extended to type graphs and later to typed graphs.

**Definition 6.1 (graph).** A graph $G = (V,E,\text{src},\text{tgt})$ consists of a finite set $V$ of nodes, a finite set $E$ of edges and two functions $\text{src},\text{tgt} : E \rightarrow V$ specifying source and target of an edge, respectively. A type graph $\mathcal{T}G$ is a graph with unique labels for all nodes and edges. For simplicity, we avoid an additional label function in favor of identifying variable names with labels. For multiple graphs we refer to the node set $V$ of a graph $G$ as $V_G$ and analogously for edge sets and the src,tgt functions. We further define the degree of a node as $\deg : V \rightarrow \mathbb{N}, v \mapsto \#\{e \in E \mid \text{src}(e) = v\} + \#\{e \in E \mid \text{tgt}(e) = v\}$. As there may be multiple graphs containing the same node, we use $\deg_G(v)$ to specify the degree of a node $v$ with respect to the graph $G$. When the context graph is clear the subscript is omitted.

Formally, type information for a graph is given by a graph morphism between the type graph and the typed graph, as defined below.

**Definition 6.2 (graph morphism, typed graph).** Given graphs with $G_i = (V_i,E_i,\text{src}_i,\text{tgt}_i)$ for $i = 1,2$ a graph morphism $f : G_1 \rightarrow G_2, f = (f_V,f_E)$ consists of two functions $f_V : V_1 \rightarrow V_2$
6. Graph Transformation Systems

and \( f_E : E_1 \to E_2 \) that preserve the source and target functions, i.e. \( f_V \circ \mathrm{src}_1 = \mathrm{src}_2 \circ f_E \) and \( f_V \circ \mathrm{tgt}_1 = \mathrm{tgt}_2 \circ f_E \).

A graph morphism \( f \) is injective (or surjective) if both functions \( f_V, f_E \) are injective (or surjective, respectively); \( f \) is called isomorphic if \( f_V(V_1) \subseteq V_1 \) and \( f_E(E_1) \subseteq E_1 \). When the context is clear, we simply refer to graph morphisms as morphisms. When there exists a graph isomorphism between two graphs \( G_1 \) and \( G_2 \), we denote this by \( G_1 \cong G_2 \).

A typed graph \( G \) is a tuple \( (V,E, \mathrm{src}, \mathrm{tgt}, \mathrm{type}, TG) \) where \( (V,E, \mathrm{src}, \mathrm{tgt}) \) is a graph, \( TG \) a type graph, and \( \mathrm{type} \) a graph morphism with \( \mathrm{type} = (\mathrm{type}_V, \mathrm{type}_E) \) and \( \mathrm{type}_V : V \to TG_V, \mathrm{type}_E : E \to TG_E \).

For a typed graph \( G = (V,E, \mathrm{src}, \mathrm{tgt}, \mathrm{type}, TG) \) we define a subgraph \( H \) as a typed graph \( (V',E', \mathrm{src}', \mathrm{tgt}', \mathrm{type}', TG) \) such that \( V' \subseteq V \land E' \subseteq E \land \mathrm{src}' = \mathrm{src} |_{E'} \land \mathrm{tgt}' = \mathrm{tgt} |_{E'} \land \mathrm{type}'_V = \mathrm{type}_V |_{V'} \land \mathrm{type}'_E = \mathrm{type}_E |_{E'} \) with \( \forall e \in E'. \mathrm{src}'(e) \in V' \land \mathrm{tgt}'(e) \in V' \).

**Example 6.3.** Figure II.1 shows an example for a type graph and a corresponding typed graph. The type graph at the top defines two types of nodes: processes and resources. Furthermore, it defines use edges going from processes to resources. The typed graph is one possible instance of a graph modeling processes and resources being used by those processes. The type graph morphism is represented by the dotted lines, showing how the nodes are typed as processes or resources, respectively.

As stated in the beginning, graph transformation systems are a rule-based approach. Hence, we next define what a rule in the GTS context consists of.

**Definition 6.4 (GTS, Rule).** A graph transformation system (GTS) \( S = (TG, R) \) is a tuple consisting of a type graph \( TG \) and a set \( R \) of graph production rules. A graph production rule – simply called rule or production rule if the context is clear – is a tuple \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) of graphs \( L, K, \) and \( R \) with inclusion morphisms \( l : K \to L \) and \( r : K \to R \).

We distinguish two kinds of typed graphs: rule graphs and host graphs. Rule graphs are the graphs \( L, K, R \) of a graph production rule \( p \) and host graphs are graphs to which graph production rules are applied. In the following section, we introduce the semantics of these rule applications.
6.2 Semantics

This work is based on the double-pushout approach (DPO) as defined in [Ehrig et al., 2006]. Most notably, we require a match morphism \( m : L \to G \) to apply a rule \( p \) to a typed host graph \( G \). The transformation yielding the typed graph \( H \) is written as \( G \xrightarrow{p,m} H \). As usual, \( \Rightarrow^* \) denotes the reflexive-transitive closure of \( \Rightarrow \). There exist two different categorical constructions for \( H \), which we present in the following subsections.

6.2.1 Double-Pushout Approach

The first approach is referred to as the double-pushout (DPO) approach and is described in more detail in [Ehrig et al., 2006]. In this approach, \( H \) is given mathematically by constructing \( D \) as shown in Figure II.2, such that (1) and (2) are pushouts in the category of typed graphs. Intuitively, the graph \( L \) is matched to a subgraph of \( G \) and its occurrence in \( G \) is then replaced by the graph \( R \). The intermediate graph \( K \) is the context graph, which contains the nodes and edges in both \( L \) and \( R \), i.e. all nodes and edges matched to \( K \) remain during the transformation.

A graph production rule \( p \) can only be applied to a host graph \( G \) if the following gluing condition is satisfied. In fact, Ehrig et al. [2006] show, that \( D \) and the pushout (1) exist if and only if this gluing condition is satisfied. It is based on the following three sets [Ehrig et al., 2006]:

- \( \text{gluing points: } GP = l(K) \)
- \( \text{identification points: } IP = \{ v \in V_L \mid \exists w \in V_L, w \neq v : m(v) = m(w) \} \cup \{ e \in E_L \mid \exists f \in E_L, e \neq f : m(e) = m(f) \} \)
- \( \text{dangling points: } DP = \{ v \in V_L \mid \exists e \in E_G \setminus m(E_L) : \text{src}_G(e) = m(v) \lor \text{tgt}_G(e) = m(v) \} \)

Hence, gluing points refer to those elements in the left-hand graph \( L \) that correspond to elements from the context graph, i.e. those that are not removed during rule application. Identification points occur for non-injective match-morphisms \( m \), when multiple elements of \( L \) are matched to the same element in \( G \). Finally, dangling points are nodes with adjacent edges, where the node’s removal by the rule application would leave edges dangling.

**Definition 6.5 (gluing condition).** The gluing condition is defined as \( IP \cup DP \subseteq GP \).

If the gluing condition is satisfied for a rule \( p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R) \) the application of the rule consists of transforming \( G \) into \( H \) by performing the construction described above. An implementation-oriented interpretation of a rule application is that all nodes and edges in \( m(L \setminus l(K)) \) are removed from \( G \) to create \( D = (G \setminus m(L)) \cup m(l(K)) \) and then all nodes and edges in \( n(R \setminus r(K)) \) are added to create \( H = D \cup n(R \setminus r(K)) \).
Example 6.6. Figure II.3 shows two graph production rules in a shorthand notation that defines the morphisms \( l \) and \( r \) implicitly by the labels of the nodes which are mapped onto each other. The resulting graph transformation system is implicitly defined over the simple type graph consisting only of a single node with a loop, depicted in Figure II.4. The two rules constitute a graph transformation system for detecting cyclic lists. The basic idea of the unlink rule is to remove intermediate nodes of the list, while the twoloop rule replaces the cyclic list consisting of two nodes by a single node with a loop. Application of the twoloop rule requires that no additional edges are adjacent to the removed node. Such dangling edges are discussed in more detail in Section 17.

To detect if a host graph is a cyclic list, the GTS is applied to the host graph until exhaustion, i.e. until no rule is applicable anymore. The initial host graph then is a cyclic list if and only if the final graph consists of a single node with a loop (cf. [Bakewell et al., 2003]).

In general, the match morphism \( m \) can be non-injective. However, for the remainder of this work we only consider injective match morphisms, which have the advantage that the set \( IP \) of identification points is guaranteed to be \( \emptyset \). Furthermore, non-injective match morphisms can be simulated as follows: given a rule \( p = (L \xrightarrow{\leftarrow} K \xrightarrow{\rightarrow} R) \) and a non-injective match morphism \( m \) it holds \( \forall v, w \in V_L, v \neq w \) with \( m(v) = m(w) \) that the rule is only applicable, if \( v, w \in l(V_K) \), i.e. only nodes which are not removed by the rule application are allowed to be matched non-injectively – otherwise \( IP \not\subseteq GP \). Therefore, it is possible to add another rule \( p' \) which is derived from \( p \) by merging the nodes \( v \) and \( w \) into a node \( v_w \) in all three graphs of the rule. Thus, the non-injective matching with \( m(v) = m(w) \) can be simulated by injectively matching \( v_w \) to \( m(v_w) \), where \( m(v_w) \) is the same node in \( G \) as \( m(v) \). The same argumentation holds for edges, analogously. Therefore, we can restrict ourselves to injective match morphisms by extending the set of rules with new rules for all possible merges of nodes and edges in the graph \( K \). This simplifies the generic gluing condition to \( DP \subseteq GP \).

Finally, we require the following definition of the track morphism [Plump, 1995]. Intuitively, the track morphism is defined for a node or edge, if it is not removed by the considered rule applications.

---

6. Graph Transformation Systems

![unlink:](image1)

**Figure II.3:** Graph transformation system for recognizing cyclic lists

![twoloop:](image2)

**Figure II.4:** Simple type graph consisting of a node and edge
Definition 6.7 (track morphism). Given $G \Rightarrow H$ the track morphism $\text{tr}_{G \Rightarrow H} : G \to H$ is the partial graph morphism defined by

$$\text{tr}_{G \Rightarrow H}(x) = \begin{cases} 
  g(f^{-1}(x)) & \text{if } x \in f(D), \\
  \text{undefined} & \text{otherwise}.
\end{cases}$$

Here $f : D \to G$ and $g : D \to H$ are the morphisms in the lower row of the pushout (1) in Figure II.2 and $f^{-1} : f(D) \to D$ maps each item $f(x)$ to $x$.

The track morphism of a derivation $\Delta : G_0 \Rightarrow^* G_n$ is defined by $\text{tr}_\Delta = \text{id}_{G_0}$ if $n = 0$ and $\text{tr}_\Delta = \text{tr}_{G_1} \cdot \ast G_n \circ \text{tr}_{G_0} \Rightarrow G_1$ otherwise, where $\text{id}_{G_0}$ is the identity morphism on $G_0$.

6.2.2 Single-Pushout Approach

The encoding of GTSs in CHR, as presented in Section 17, is based on the double-pushout approach for graph transformation systems. A related graph rewriting mechanism, the single-pushout approach, was introduced by Löwe [1993]. Instead of demanding two pushouts, as in Figure II.2, it defines rewriting over a category of partial graph morphisms, hence only a single-pushout construction is used.

Intuitively, this results in a different behavior with respect to dangling edges: While the double-pushout approach prohibits a rule application in case a dangling edge would remain, the single-pushout approach removes all dangling edges instead. In [Löwe and Müller, 1993] the authors investigate confluence for single-pushout graph rewriting. In particular, the critical pair analysis is shown to be only a sufficient criterion as well, not a necessary one.

In our work, we focus on the double-pushout approach, because its behavior with respect to dangling edges can be represented more naturally in CHR. Encoding the single-pushout approach in CHR would require that dangling edges get removed by a rule application, although by their definition, we do not have knowledge of them in the actual rule application. Hence, additional clean-up rules would be needed and we would have to ensure correct scheduling, as the clean-up rules have to be executed before new rule applications are tried in order to have consistent degrees.

7 Program Analysis

Program analysis is an important discipline in computer science. Through it we determine properties of programs, like termination, confluence or time complexity, which are relevant to their intended usage. The declarative rooting of CHR, as well as the category theoretical foundation of GTSs, facilitate sound program analysis. Important results have been achieved for the analysis of termination, confluence, completion, modularity, program equivalence, time and space complexity, among others.

This section presents an overview of important program analysis methods available for Constraint Handling Rules and graph transformation systems. In this work, we focus on confluence and program equivalence, which are discussed in detail in Sections 14 and 15.

We will not present formal definitions and results in this section. For the benefit of a unified presentation, we introduce these after Chapter III details the operational semantics of CHR used throughout the main body of this work. Adapted formulations of definitions and results on confluence in CHR are given in Section 14.1, and respectively in Section 15.1 for program equivalence.
We will first discuss results for properties other than confluence and program equivalence in Section 7.1, before we present results available in the literature for confluence in Section 7.2 and for program equivalence in Section 7.3.

7.1 Overview

This section provides an overview of the literature with regards to program analysis methods for properties other than confluence and program equivalence. We begin with results available for Constraint Handling Rules, before we present corresponding results for graph transformation systems.

Termination As CHR is Turing-complete [Sneyers et al., 2005], the termination problem is undecidable in general. An early termination analysis method was derived from results for term rewriting systems: Given a mapping \( \varphi \) from states into \( \mathbb{N} \), often called measure function, it suffices to prove that each rule application results in a state with a lower mapped number (cf. [Baader and Nipkow, 1998]). This technique was first adapted for termination analysis of CHR programs by Frühwirth [2000].

The results in [Frühwirth, 2000] apply only to CHR programs without propagation rules. A different approach, based on conditions imposed on the addition of constraints, has been introduced by Voets et al. [2007]. It specifically supports programs with propagation rules. The work of Pilozzi and De Schreye [2008] improves upon this: They link size-decreases of a different representation of CHR states to termination, which provides a strictly more powerful condition than both previous approaches.

Completion Completion is a method for modifying a non-confluent program such that it becomes confluent. Early results for term rewriting systems date back to the work from Knuth and Bendix [1970], which has been adapted to CHR by Abdennadher and Frühwirth [1998].

Modularity Multiple CHR programs can be considered as modules to be combined in two different ways. A flat union simply merges all rules into a single CHR program. This kind of modularity has been investigated by Abdennadher and Frühwirth [2003] and it has been shown, that in general confluence and termination of individual programs is not preserved for their union.

The second possibility is a hierarchical approach, which allows CHR constraints from one program to be reused in other programs as built-in constraints. This requires that the implication of such a constraint can be checked. Schrijvers et al. [2006] investigated checking them automatically, whereas Fages et al. [2008] proposed user-definable rules for implication checking via ask and tell constraints.

Complexity Frühwirth was the first to investigate automatic time complexity analysis for CHR programs in [Frühwirth, 2002] and [Frühwirth, 2001]. However, this analysis is based on a naive CHR compiler without support for important optimizations, hence, the approach yields only weak upper boundaries.

As CHR rule heads can contain multiple constraints, it is incessant to optimize the code required for matching these head constraints to constraints in the store. A naive approach, which would try to match each head constraint with each constraint in the
store, leads to exorbitantly slow runtimes. Therefore, CHR research produced a significant body of work on optimizations of CHR execution. A summary of the techniques developed up until 2005 is available in [Schrijvers, 2005]. As complexity analysis is outside the scope of this work, we will only list the numerous other improvements that have been published since: [Schrijvers and Frühwirth, 2006, De Koninck and Sneyers, 2007, Sarna-Starosta and Ramakrishnan, 2007, Wuille et al., 2007, De Koninck et al., 2008, Sarna-Starosta and Schrijvers, 2008, Van Weert, 2008, De Koninck, 2009, Sarna-Starosta and Schrijvers, 2009].

Computational Power A notable line of research was the investigation of the computational power of CHR, which culminated in the PhD thesis from Sneyers [2008a]. This work assumes a CHR compiler that supports several important optimization techniques, which leads to the following significant result: any algorithm can be implemented in CHR in optimal time and space complexity.

After the proof that CHR in its general form is Turing-complete [Sneyers et al., 2005], restrictions of CHR, for example, to single-headed rules or limited built-in theories, have been investigated. This lead to the discovery of several Turing-complete subclasses of CHR [Sneyers, 2008b, Sneyers et al., 2009b, Gabbrielli et al., 2010, Mauro et al., 2010].

Program analysis has been an active CHR research topic for the past two decades. In contrast, there are not as many results available for graph transformation systems, where we currently witness increased interest in their analysis.

Termination As with all rewriting systems, termination can be shown via a measure function. However, this approach is often not powerful enough. Termination analysis in the GTS context appears to provide different approaches from CHR. For example, Ehrig et al. [2006] presents a criterion specifically for layered graph transformation systems, in which rule applications are sorted into individual layers. Each layer is exhaustively applied, before continuing with the next layer.

Another interesting approach is given by Levendovszky et al. [2007], who substitutes sequential rule applications by their composition. He then investigates, if for infinite rule sequences the left-hand side of their composition tends to infinity, which implies termination, because the initial graph only contains finitely many elements.

Modularity In the GTS context, graph transformation units appear to be the most prominent means for modularity. A graph transformation unit contains a set of rules, as well as a description of initial and terminal graphs and a control condition. A survey on them is available by Kreowski and Kuske [1999]. In contrast to CHR research, however, we are not aware of investigations of program properties with respect to modularity.

Complexity Complexity of graph transformation systems is inherently problematic, because finding a match for a left-hand side of a rule in the host graph corresponds to the subgraph isomorphism problem. This problem has already been shown to be NP-complete by Cook [1971] in his seminal paper.

In practice, however, left-hand sides are fixed and have finite size, such that the exponential argument cannot increase in an unlimited fashion. CHR itself suffers from a similar problem in the choice of partner constraints. However, indexing is an important...
optimizing technique in CHR, which allows us to retrieve partner constraints efficiently. A related approach has been investigated for GTS by Dodds and Plump [2006].

The core idea of that work is to consider GTSs with a special marker node. Every left-hand side of a rule then needs to include this marker node and consist of a connected graph. The same marker node has to be present in the initial graph, which then essentially allows for constant time matching. The underlying idea is similar to indexing, because the marker node uniquely identifies its neighbors, analogously to a CHR constraint’s argument identifying another constraint.

7.2 Confluence

In this section, we discuss the so-called confluence property, its importance, and existing results related to confluence of CHR programs and graph transformation systems. Confluence is a property that is relevant to all non-deterministic rewrite systems. Non-determinism allows for a state to be rewritten into two or more possibly different states, which naturally prompts the following question: for a given starting state $S_0$ and diverging computations, is it always possible to join the computations again? More generally, we are interested in whether this property holds for all possible starting states. If it does, the corresponding rewrite system is called confluent. Figure II.5 visualizes the confluence property, which is also referred to as the diamond property due to the shape of this depiction. Confluence has been investigated for all kinds of non-deterministic rewrite systems by now, but originated from the investigation of term rewriting systems. [Baader and Nipkow, 1998, Ohlebusch, 2002] provide a summary of the results related to confluence of term rewriting systems.

The importance of confluence stems from observing non-deterministic systems: In a confluent rewrite system, it is sufficient to consider a single rewriting path, because the final state will be uniquely determined by it. This allows us to consider different rewriting strategies and even select the most suitable one at any intermediate rewriting step. Furthermore, confluence implies straightforward parallelization possibilities: as the order of rewriting steps can be modified without affecting the result, we can rewrite independent parts of a state in parallel. Finally, a confluent rewriting system facilitates program analysis. Hence, it is not uncommon that a program analysis method requires confluent programs (cf. f.ex. [Abdenadher and Frühwirth, 1999]).

In order to decide confluence of rewrite systems, a general result based on the investigation of so-called critical pairs has been given by Huet [1980] in his seminal article. In that approach, all syntactical overlaps of rule pairs are considered, which lead to pairs of states that are then checked for confluence. The resulting property is called local confluence and if it can be
verified for all rule pairs and the rewrite system is known to be terminating, then the lemma by Newman [1942] implies confluence of the whole system.

### 7.2.1 Confluence for Constraint Handling Rules

For Constraint Handling Rules, confluence was first investigated by Meuss [1996] and Abdennadher et al. [1996]. Abdennadher continued this research [Abdennadher, 1997, Abdennadher et al., 1999], which culminated in his Habilitationsschrift [Abdennadher, 2001]. His work adapted existing results from term rewriting systems to the context of CHR, which differs significantly due to its support for built-in constraints and propagation rules. In particular, he managed to formulate a critical pair condition, which permits us to decide confluence for terminating CHR programs. This work was further improved upon by Raiser and Tacchella [2007] with a criterion for non-terminating CHR programs and by Haemmerlé and Fages [2007], who reduced the number of critical pairs that need to be investigated.

Furthermore, Duck et al. [2007] introduced observable confluence, which provides a generalized critical pair condition. The syntactical construction of critical pairs often leads to some, which are known to never occur during a normal execution of the application. Hence, in those cases non-confluence caused by an irrelevant critical pair is irrelevant. Observable confluence therefore relies on an invariant to determine, which of the critical pairs are relevant.

We will formally introduce the necessary definitions for confluence in CHR in Section 14, based on our novel operational semantics defined in Chapter III. Furthermore, our definitions generalize some of the above mentioned works, however, we will discuss the necessary specializations to retrieve the original works.

### 7.2.2 Confluence for Graph Transformation Systems

The investigation of confluence for graph transformation systems was mainly undertaken by Plump, beginning with [Plump, 1993], where he shows that the critical pair condition from term rewriting systems is not applicable to graph transformation systems. In [Plump, 2005] he gives a stronger result, namely, that confluence is undecidable for a terminating GTS. This result is noteworthy, because it separates GTSs from other rewriting systems. In particular, it makes it interesting for us to investigate the effect of applying the decidable confluence test on a terminating graph transformation system encoded in CHR (cf. Section 19). Plump’s results are based on the DPO approach, whereas Löwe [1993] verifies for the SPO approach that the critical pair condition is also only a sufficient criterion for confluence.

Next, we present the formal results for confluence of graph transformation systems. In contrast to the CHR context, this thesis makes no changes to those existing definitions and results. Instead, we will investigate in Section 19 how these existing characterizations of confluence apply to a GTS encoded in CHR.

**Definition 7.1 (GTS Confluence).** A GTS is called confluent if, for all typed graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$, there is a typed graph $X$ together with typed graph transformations $H_1 \Rightarrow X$ and $H_2 \Rightarrow X$. Local confluence means that this property holds for all pairs of direct typed graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$ [Ehrig et al., 2006].

Newman’s general result for rewriting systems implies that it is sufficient to consider local confluence for terminating graph transformation systems. To verify local confluence, one
typically studies critical pairs and their joinability, according to the following definition based on [Ehrig et al., 2006, Plump, 2005].

**Definition 7.2** (Joinability of Critical GTS Pair). Let $r_1 = (L_1 \leftarrow K_1 \rightarrow R_1), r_2 = (L_2 \leftarrow K_2 \rightarrow R_2)$ be two GTS rules. A pair $P_1 \xleftarrow{m_1} G \xrightarrow{m_2} P_2$ of direct typed graph transformations is called a critical GTS pair if it is parallel dependent, and minimal in the sense that the pair $(m_1, m_2)$ of matches $m_1: L_1 \rightarrow G$ and $m_2: L_2 \rightarrow G$ is jointly surjective.

A pair $P_1 \xleftarrow{m_1} G \xrightarrow{m_2} P_2$ of direct typed graph transformations is called parallel independent if $m_1(L_1) \cap m_2(L_2) \subseteq m_1(K_1) \cap m_2(K_2)$, otherwise it is called parallel dependent.

A critical GTS pair $P_1 \xleftarrow{m_1} G \xrightarrow{m_2} P_2$ is called joinable if there exist typed graphs $X_1, X_2$ together with typed graph transformations $P_1 \Rightarrow X_1 \simeq X_2 \Leftarrow P_2$. It is strongly joinable if there is an isomorphism $f: X_1 \rightarrow X_2$ such that for each node $v$, for which $\text{tr}_G \Rightarrow P_1(v)$ and $\text{tr}_G \Rightarrow P_2(v)$ are defined, the following holds:

1. $\text{tr}_G \Rightarrow P_1(v)$ and $\text{tr}_G \Rightarrow P_2(v)$ are defined and
2. $f_V(\text{tr}_G \Rightarrow P_1(v)) = \text{tr}_G \Rightarrow P_2(v)$

Plump [2005] provided the negative result for the confluence decision problem, even for terminating graph transformation systems.

**Theorem 1** (Undecidability of GTS Confluence [Plump, 2005]). Confluence of a terminating graph transformation system is undecidable.

*Proof.* By a reduction to the Post Correspondence Problem (see [Plump, 2005]).

Nevertheless, the original critical-pair-based approach proposed by Huet [1980] can be adapted to the GTS context, which yields the following sufficient criterion for confluence.

**Theorem 2** (Sufficient Criterion for GTS Confluence [Plump, 2005]). A terminating graph transformation system is confluent if all its critical GTS pairs are strongly joinable.

*Proof.* see [Plump, 2005].

In Section 19, we show that proving confluence of a CHR program that encodes a terminating GTS, corresponds to applying the above theorem, and hence, is a sufficient criterion for confluence of the encoded GTS.

### 7.3 Program Equivalence

When should we consider two programs equivalent? – This is a fundamental question for programming languages, and hence, the literature provides an abundance of works on this topic. However, when it comes to equivalence of rewriting systems, only a fraction of these remain applicable. And if one considers logic-based languages the results are even sparser, up to the point that for CHR, a logic-based rewriting system, only one general result has been published so far [Abdennadher and Frühwirth, 1999]. For this reason, the remainder of this section discusses related works from the point of view of our intended kind of program equivalence.
Bisimilarity Investigations of program equivalence of rewriting systems have mostly taken place in the context of process calculi throughout the last decade. The concept of bisimilarity [Milner, 1989] was identified as a suitable means of comparing observable behavior of processes. It is based on labeled transition systems and observing the process transitions with respect to their labels. Bisimilarity is primarily targeted at non-terminating processes and requires a strong correspondence between states in these processes.

Weak bisimulation [Milner, 1989] weakens this correspondence by ignoring intermediate transitions by means of so-called internal transitions. However, weak bisimulation still requires that for a state reached by internal transitions a corresponding state reachable by internal transitions exists in the other program. Hence, there still needs to be a correspondence between intermediate states in both programs, not only final states.

For bisimilarity the problem of automating bisimulation proofs gave rise to so-called up-to techniques [Hirschkoff, 2001]. These techniques are based on evolving the processes such that the size of relations is reduced.

n-Equivalence In this work however, we will not share the setting of non-terminating processes used by bisimilarity research. Instead we want to compare program equivalence with respect to the equivalence of computed answers, independently of the way these were reached.

Mohan [1991] defined n-equivalence as computing the same normal form relations, i.e. reaching the same final states. He considered different types of term-based rewriting systems.

His results are not directly applicable to CHR, because in CHR normal forms may be syntactically different. However, our proposed equivalence-based operational semantics (cf. Section 8) ensures that we get an equivalence class containing all possible syntactical representations as a normal form. Hence, we can take the core idea of Mohan and adapt it to CHR with the results given in Section 15.3.

Restricted Domains We also want to be able to have a flexible program equivalence test that is able to account for implicit assumptions about a program. One such typical assumption is the knowledge that both programs will only be executed on certain kinds of input.

Toyama [1985] considered equivalence for term rewriting systems in a restricted domain, which intuitively corresponds to limiting input possibilities to more closely reflect the intended usage of programs. However, this approach is not general enough and we aim for an invariant-based approach, similar to observable confluence [Duck et al., 2007].

Semantic-Based Equivalences The work in Gabrrielli et al. [1995] is concerned with equivalences of observable behavior of programs. It extends Maher [1988] by considering relationships between equivalences derived from semantics that are based, for example, on computed answer substitutions. However both works are not concerned with testing equivalence.

Operational Equivalence Abdennadher and Frühwirth [1999] showed that equivalence for terminating and confluent programs in CHR, and hence multiset rewriting can be decided for a large class of programs. His work is based on so-called operational equiva-
7. Program Analysis

lence, which includes ideas from both bisimilarity and n-equivalence in that it requires a close correspondence between states of the two systems and concentrates on normal forms. However, operational equivalence itself is often too strict. This was already apparent in [Abdennadher and Frühwirth, 1999], and therefore, loosened to operational c-equivalence for a single constraint symbol c.

It is the only available result for program equivalence in CHR and we will present it in more detail in Section 15.1.

Behavioral Equivalence To the best of our knowledge, there are no results for program equivalence of GTSs based on normal forms. However, there exists an orthogonal notion of behavioral equivalence for graph transformation systems [Rangel et al., 2008]. Behavioral equivalence investigates the behavior of models, i.e. host graphs, which are transformed into new models while preserving behavior as given by a semantics based on another graph transformation system.

Our results on program equivalence analysis will be presented in Section 15. In particular, we will extend the work by Abdennadher and Frühwirth [1999] to make the approach more generic and thus extend the test to show equivalence of a greater set of CHR programs. Furthermore, we will transfer the results from Abdennadher and Frühwirth [1999] to the context of graph transformation systems.
Chapter III

A Complete and Terminating Operational Semantics for Constraint Handling Rules

Each problem that I solved became a rule, which served afterwards to solve other problems.
— René Descartes (1596–1650), Mathematician

While CHR is known as a language that combines efficiency with declarativity, publications in the field display a tendency to favor one of these aspects over the other. We observe a spectrum of research directions ranging from the analytical to the pragmatic.

On the analytical end of the spectrum, emphasis is put on CHR as a mathematical formalism, declarativity, and the understanding of its logical foundations and theoretical properties. Several formalizations of the operational semantics, found in [Frühwirth, 1998] and [Frühwirth and Abdennadher, 2003], belong to this side of the spectrum.

On the downside, these operational semantics are detached from practical implementation in that they are oblivious to questions of efficiency and termination. Particularly, the class of rules called propagation rules causes trivial non-termination in both of them. Hence, it is safe to say that the existing analytical formalizations of the operational semantics lack a terminating execution model.

Example 7.3 (Lack of Termination with $\omega_{va}$). Consider the following propagation rule, which is part of a lower-equal solver given in [Frühwirth, 2009] and corresponds to the transitivity relation.

$$\text{leq}(A, B), \text{leq}(B, C) \rightarrow \text{leq}(A, C)$$

The main idea behind this rule is that in some cases creating an additional leq-constraint may help to further simplify the store. However, under the very abstract operational semantics $\omega_{va}$, we witness the following infinite derivation.

$$\begin{align*}
\langle \text{leq}(A, B) \land \text{leq}(B, C) \rangle \\
\Rightarrow_{va} \langle \text{leq}(A, B) \land \text{leq}(B, C) \land \text{leq}(A, C) \rangle \\
\Rightarrow_{va} \langle \text{leq}(A, B) \land \text{leq}(B, C) \land \text{leq}(A, C) \land \text{leq}(A, C) \rangle \\
\Rightarrow_{va}^* \langle \text{leq}(A, B) \land \text{leq}(B, C) \land \text{leq}(A, C) \land \text{leq}(A, C) \land \text{leq}(A, C) \land \ldots \rangle \\
\Rightarrow_{va}^* \ldots
\end{align*}$$

This contrasts with most work on the pragmatic side of the spectrum, which emphasizes practical implementation and efficiency over formal reasoning. It originates with Abdennadher
[1997], who proposed a token-based approach in order to avoid trivial non-termination: Every
propagation rule is applicable only once to a specific combination of constraints. Thus, a
terminating execution model for the full segment of CHR is provided, which however, lacks
completeness.

**Example 7.4** (Lack of Completeness with $\omega_t$). Consider the following two rules and their
execution under the theoretical operational semantics $\omega_t$.

\[
\begin{align*}
\text{r}_1 & \@ a \rightarrow b \\
\text{r}_2 & \@ b, b \leftrightarrow c
\end{align*}
\]

We know that in the abstract definition of CHR it should be possible with these rules to derive
d c-constraint, when starting with a single a-constraint. Clearly, there exists a derivation
under $\omega_{va}$ to achieve this, however, we cannot ever reach a c-constraint under $\omega_t$, as the
following derivation shows.

\[
\begin{align*}
\langle a; \emptyset; T; \emptyset \rangle_0 & \rightarrow_t \langle \emptyset; a\#0; T; \emptyset \rangle_1^0 \\
\rightarrow_t \langle b; a\#0; T; (r_1, 0) \rangle_1^0 & \rightarrow_t \langle \emptyset; a\#0, b\#1; T; (r_1, 0) \rangle_2^0 \\
& \not\rightarrow_t
\end{align*}
\]

The final state demonstrates the essence of the effect of propagation tokens: Termination of
rule $r_1$ for the identified constraint $a\#0$ is ensured at the cost of completeness.

In this chapter, we present the development of the operational semantics $\omega_t$, which provides
a complete and terminating execution model for CHR. We begin in Section 8 with an in-
vestigation of equivalence of CHR states. It turns out, that an axiomatic formulation
of state equivalence tremendously simplifies the formulation of analytical operational semantics
of CHR. In that section, we hence introduce $\omega_e$, which is defined as a rewriting system of
equivalence classes and is compatible with $\omega_{va}$ [Frühwirth, 2009].

Next, Section 9 introduces so-called persistent constraints. They enable us to extend $\omega_e$ into
the operational semantics $\omega_t$, which provides a complete and terminating execution model.
Building on the equivalence-based formulations of the operational semantics, we present a
merge operator in Section 10, which is a useful tool in program analysis. Then, in Section 11,
we discuss the differences between $\omega_t$ and existing operational semantics and present an im-
plementation of $\omega_t$ via a source-to-source transformation. Finally, we discuss related work
and future research paths in Section 12.

# 8 Equivalence-based Operational Semantics

While equivalence of states is an elementary concept in Constraint Handling Rules (CHR),
the community has never agreed on a standard definition of that concept up to now. A
plethora of definitions of state equivalence has been introduced in various areas of application
(cf. Section 8.1.1). For example, the operational equivalence algorithm [Abdennadher and
Frühwirth, 1999] compares two final states of different programs for equivalence. State equiva-
rence is the basis for invariants such as in [Raiser and Frühwirth, 2009c]. Several definitions
have been introduced in the context of confluence considerations.
As the various authors had different intentions, the resulting definitions of state equivalence vary considerably. There is a general agreement that, from an operational point of view, any notion of state equivalence should be compliant with rule applications, i.e. for equivalent states the same rules are applicable and lead to equivalent results. However, this property has never been proven for any of the previously proposed definitions. Another general agreement is that from a declarative point of view the logical reading of equivalent states should also be equivalent.

Our aim is therefore to develop a definition of state equivalence that satisfies both the operational and the declarative view. Instead of defining a notion of state equivalence for a fixed problem setting, we intend a notion for which these generally agreed-upon properties hold. By construction, our definition of state equivalence then is compliant with rule application and the logical reading of states. Thus, it becomes a generic proof technique that can be applied to specific problems with the additional knowledge that the above-mentioned properties are satisfied.

In Section 8.1, we investigate equivalence of CHR states and provide an axiomatic definition, as well as a decidable criterion, for it. Then, Section 8.2 presents the operational semantics $\omega_e$ based on rewriting of equivalence classes.

8.1 State Equivalence

In this section, we first justify a set of desirable properties for a general notion of state equivalence in Section 8.1.1 and present them in the form of example cases. Then, we give a concise overview of the existing definitions of state equivalence and compare their behavior with respect to these example cases. We show that none of the existing definitions satisfies all of the example cases.

Next, Section 8.1.2 introduces an axiomatic definition of state equivalence along with several useful properties following from that definition. We show that it satisfies all the previously investigated example cases. Finally, we present a necessary, sufficient, and decidable criterion for determining equivalence of states in Section 8.1.3.

8.1.1 Existing State Equivalence Definitions

In this section, we evaluate existing definitions of state equivalence and postulate desirable properties of an equivalence relation over CHR states. To this end, we present several prototypical example cases of equivalent and non-equivalent CHR states. We concisely introduce the different notions of state equivalence that have been proposed so far, before we investigate how these notions apply to our example states.

In favor of a unified presentation, we will use the definition of $\omega_e$ states throughout this section. It clearly separates the three components that each have to be treated differently by state equivalence. We accordingly adapt existing definitions and results to this definition.

**Definition 8.1 ($\omega_e$ State).** A $\omega_e$ state is a tuple

$$\langle G; B; V \rangle.$$  

- The goal $G$ is a multiset of CHR constraints.
- The built-in constraint store $B$ is a conjunction of built-in constraints.
8. Equivalence-based Operational Semantics

- \( \forall \) is a set of global variables.

We use \( \sigma, \sigma_0, \sigma_1, \ldots \) to denote \( \omega_e \) states and \( \Sigma_e \) to denote the set of all \( \omega_e \) states. We denote the state \( (\emptyset; \top; \emptyset) \) as \( \sigma_0 \).

We sometimes also use \( \Sigma \) to denote the set of all states, in cases where multiple operational semantics may apply. For example, many results of our proposed operational semantics presented in Section 9 equally apply to \( \Sigma_e \).

**Examples of Equivalences of CHR States**  Let us now consider the following examples of equivalent and non-equivalent states to highlight the differences between existing definitions of state equivalence. The relation \( \equiv \) is used here as a generic equivalence relation with our desired properties. We will refer to our equivalence relations, defined later, as \( \equiv_e \) and \( \equiv! \) instead.

\[
\begin{align*}
\langle c(X); \top; \emptyset \rangle & \equiv \langle c(Y); \top; \emptyset \rangle \quad (\text{III.1}) \\
\langle c(X); X = 0; \{X\} \rangle & \equiv \langle c(0); X = 0; \{X\} \rangle \quad (\text{III.2}) \\
\langle \top; X \geq 0 \land X \leq 0 \land Y = 0; \{X\} \rangle & \equiv \langle \top; X = 0; \{X\} \rangle \quad (\text{III.3}) \\
\langle c(0); \top; \{X\} \rangle & \equiv \langle c(0); \top; \emptyset \rangle \quad (\text{III.4}) \\
\langle c(X); \top; \{X\} \rangle & \not\equiv \langle c(Y); \top; \{Y\} \rangle \quad (\text{III.5})
\end{align*}
\]

The equivalences (III.1)-(III.3) are motivated by the fact that the same rules are applicable to these states. Specifically, equivalence (III.1) covers renaming of local variables, equivalence (III.2) the substitution of variables with terms, and equivalence (III.3) built-in stores that are logically equivalent under the constraint theory \( CT \). Included in equivalence (III.3), are built-in constraints over strictly local variables (\( Y = 0 \)), which may be removed due to not affecting logical equivalence in any way (\( \exists Y. Y = 0 \) is a tautology).

As the states in equivalence (III.4) have the same logical reading \( c(0) \) (cf. Definition 5.16), we require them to be equivalent. Unused global variables can practically occur, for example, when applying rule \( c(X) \Leftrightarrow c(0) \) to the state \( \langle c(X); \top; \{X\} \rangle \). Concerning non-equivalence (III.5), note that \( X, Y \) are free variables and therefore the logical readings \( c(X) \) and \( c(Y) \) (cf. Definition 5.16) are not equivalent.

**Existing Definitions**  Over the last decade, the CHR community proposed various definitions for state equivalence. The following list identifies six distinct categories of equivalence definitions in the literature:

1. Definitions based on variable renaming [Abdennadher et al., 1999, Frühwirth et al., 2002, Duck, 2005, Meister, 2008] are often as simple as stating that two states are equivalent (or variants) if they can be obtained by variable renaming only. These definitions arose from the notion of variance on terms.

2. In [Raiser and Tacchella, 2007] a definition is given that is based on renaming of local variables as well as logical equivalence of built-in stores.

3. In [Haemmerlé and Fages, 2007] a similar definition is given for arbitrary binary relations rather than for CHR states only.
Chapter III

<table>
<thead>
<tr>
<th></th>
<th>(III.1)</th>
<th>(III.2)</th>
<th>(III.3)</th>
<th>(III.4)</th>
<th>(III.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def. Type 1</td>
<td>≡</td>
<td>≠</td>
<td>≠</td>
<td>≠</td>
<td>≡</td>
</tr>
<tr>
<td>Def. Type 2</td>
<td>≡</td>
<td>≠</td>
<td>≡</td>
<td>≡</td>
<td>≠</td>
</tr>
<tr>
<td>Def. Type 3</td>
<td>≡</td>
<td>≠</td>
<td>≡</td>
<td>≡</td>
<td>≠</td>
</tr>
<tr>
<td>Def. Type 4</td>
<td>≡</td>
<td>≠</td>
<td>≡</td>
<td>≠</td>
<td>≠</td>
</tr>
<tr>
<td>Def. Type 5</td>
<td>≡</td>
<td>≡</td>
<td>≡</td>
<td>≠</td>
<td>≠</td>
</tr>
<tr>
<td>Def. Type 6</td>
<td>≠</td>
<td>≡</td>
<td>≡</td>
<td>≠</td>
<td>≠</td>
</tr>
<tr>
<td>Desired</td>
<td>≡</td>
<td>≡</td>
<td>≡</td>
<td>≡</td>
<td>≠</td>
</tr>
</tbody>
</table>

Table III.1: Comparison of Different State Equivalence Definitions

4. Duck et al. [2006] give another definition based on the refined operational semantics [Duck et al., 2004] of CHR.

5. Duck et al. [2007] – a follow-up to [Duck et al., 2006] – extends the definition with the usage of a unifier instead of variable renaming.

6. In [Abdennadher et al., 1999, Abdennadher, 2001] a normalization function is defined. While we emphasize that this definition was not targeted towards determining state equivalence, we include it in this work due to its clear structure that is similar to our proposed definition. When we talk about equivalence with respect to normalization we implicitly assume that two states are equivalent if and only if their normalizations are syntactically equivalent.

Comparison of Existing Equivalence Definitions

We have applied each of the existing definitions to each of the example cases. The results are presented in Table III.1. Each entry shows whether the two corresponding example states are considered equivalent or not, according to the definition type used in that row. The last row presents the results that we deem desirable.

Table III.1 reveals that definition types 1 to 4 do not satisfy the desired result for equivalence III.2, which is the important substitution of variables with terms. Definition type 6 also shows two differences to our desired properties. However, the definition of Duck et al. [2007] closely corresponds to the properties we would like to see in an equivalence relation. The only difference is in equivalence (III.4), which allows removal of unused global variables. In most applications this kind of equivalence may be unnecessary though, such that the definition given in [Duck et al., 2007] seems to be a viable candidate.

However, the formulation used for the equivalence relation in [Duck et al., 2007] is very involved. It includes a unifier that has to be determined and its application in proofs quickly becomes complicated by that. Furthermore, in a personal correspondence with the authors of [Duck et al., 2007], we were informed that the formulation given in their work was obtained through a trial and error method. This can also be seen from the fact that [Duck et al., 2006] used a slightly different, but faulty, formulation (cf. Table III.1, Def. Type 4).

Therefore, none of the previously published definitions of state equivalence respects all of the example cases.
8. Equivalence-based Operational Semantics

**Excursion:** Detailed Discussion of Comparison

In this excursion, we discuss in more detail, why the individual definitions do not meet our desired results.

The definitions of type 1 lack any means to handle equivalent representations of built-ins. This problem arises in its purest form in case (III.3). Furthermore, they usually neglect to consider global variables, hence, case (III.5) fails as well.

Definitions of type 2 extend the variable renaming by built-in equivalence, which allows them to correctly treat case (III.3). Straightforward consideration of global variables solves case (III.5) satisfactorily as well.

However, all definitions of types 1, 2, 3, and 4 fail to consider more complex interactions between built-in equivalence and variable renaming, such that they fail for case (III.2). In that case, we substitute a variable \( X \) for its value, if the built-ins imply that \( X = c \) for some value.

In fact, we will later see that the condition of permitting variable renaming is too weak. Case (III.2) revealed, that it cannot handle substitution and we will show that substitution and built-in equivalence actually subsume variable renaming. This is also the reason, why in our axiomatic definition of equivalence (cf. Definition 8.2) we have no axiom for variable renaming.

This problem of supporting substitutions has been discovered by Duck et al. [2006] as well and they resolved it by adapting their definition in [Duck et al., 2007], which is the definition type 5. As explained above, the definition turned out to be very involved and it still fails on case (III.4) in that it considers unused global variables as important. Intuitively, it appears to make more sense though to only care about elements relevant to the two states when determining their equivalence.

Finally, the normalization function, as it is defined in [Abdennadher, 2001] and represented by definition type 6 in Table III.1, fails to yield the same normal forms for a local variable, thus failing case (III.1).

### 8.1.2 Novel Axiomatic Definition of State Equivalence

In this section, we introduce our axiomatic definition of equivalence that satisfies all desirable properties we identified in the previous section.

**Definition 8.2** (Equivalence of \( \omega_e \) States). Equivalence between \( \omega_e \) states is the smallest equivalence relation \( \equiv_e \) over CHR states that satisfies the following conditions:

1. (Equality as Substitution)
   \[
   \langle G; X = t \land B; V \rangle \equiv_e \langle G[X/t]; X = t \land B; V \rangle
   \]

2. (Transformation of the Constraint Store) If \( CT \models \exists \bar{s}.B \leftrightarrow \exists \bar{s}'.B' \) where \( \bar{s}, \bar{s}' \) are the strictly local variables of \( B, B' \), respectively, then:
   \[
   \langle G; B; V \rangle \equiv_e \langle G; B'; V \rangle
   \]

3. (Omission of Non-Occurring Global Variables) If \( X \) is a variable that does not occur in \( G \) or \( B \) then:
   \[
   \langle G; B; \{ X \} \cup V \rangle \equiv_e \langle G; B; V \rangle
   \]
4. (Equivalence of Failed States)
\[
\langle G; \bot; V \rangle \equiv_e (G'; \bot; V')
\]

Names of local variables in CHR states are chosen non-deterministically upon execution. Hence, considering these names invariant with respect to state equivalence suggests itself. In combination, axiom 1 and axiom 2 guarantee this desired property (cf. Lemma 8.3:1). Axiom 1 and axiom 2 are furthermore invariant with respect to rule applicability and comply with logical equivalence of the logical readings. The same holds for axiom 4: On the logical level, inconsistent logical readings are of course logically equivalent. Operationally, unused global variables have no effect, so it stands to reason to consider them redundant.

Lemma 8.3 states several properties that follow from Definition 8.2.

**Lemma 8.3** (Properties of State Equivalence). The equivalence relation \( \equiv_e \) over CHR states given in Def. 8.2 has the following properties:

1. (Renaming of Local Variables) Let \( X, Y \) be variables such that \( X, Y \notin V \) and \( Y \) does not occur in \( G \) or \( B \):
\[
\langle G; B; V \rangle \equiv_e \langle G[X/Y]; B[X/Y]; V \rangle
\]

2. (Partial Substitution) Let \( G[X \mapsto t] \) be a multiset where some occurrences of \( X \) are substituted with \( t \):
\[
\langle G; X = t \land B; V \rangle \equiv_e \langle G[X \mapsto t]; X = t \land B; V \rangle
\]

3. (Logical Equivalence) If
\[
\langle G; B; V \rangle \equiv_e \langle G'; B'; V' \rangle
\]
then \( CT \models \exists \bar{y}. G \land B \leftrightarrow \exists \bar{y}'. G' \land B' \), where \( \bar{y}, \bar{y}' \) are the local variables of \( \langle G; B; V \rangle \), and \( \langle G'; B'; V' \rangle \), respectively.

Proof.

**Property 1:** By transformation of the constraint store, we have that \( \langle G; B; V \rangle \) is equivalent to \( \langle G; X = Y \land B; V \rangle \). We apply equality as substitution and get \( \langle G[X/Y]; X = Y \land B; V \rangle \) which by transformation is equivalent to \( \langle G[X/Y]; B[X/Y]; V \rangle \).

**Property 2:** By substitution, we have that both \( \langle G; B; V \rangle \) and \( \langle G[X \mapsto t]; X = t \land B; V \rangle \) are equivalent to \( \langle G[X/t]; X = t \land B; V \rangle \), because \( \equiv_e \) is symmetric and transitive.

**Property 3:** All conditions given in Def. 8.2 correspond to valid logical equivalences:

**Definition 8.2:1** preserves logical equivalence since
\[
G \land X = t \leftrightarrow G[X/t] \land X = t
\]

**Definition 8.2:2:** As \( CT \models \exists \bar{s}. B \leftrightarrow \exists \bar{s}'. B' \) and the variables in \( \bar{s}, \bar{s}' \) do not occur in \( G, G' \), we have
\[
CT \models \exists \bar{y}. B \land G \leftrightarrow \exists \bar{y}'. B' \land G
\]
**Definition 8.2:3** For a variable $X$ that does not occur in $B$ or $G$ we obviously have

$\mathcal{CT} \models \exists X. \exists \bar{y}. B \land G \leftrightarrow \exists \bar{y}. B \land G$

**Definition 8.2:4** preserves logical equivalence due to the *ex falso quodlibet* property. As logical equivalence is reflexive, transitive, and symmetric, Prop. 3 holds.

**Example 8.4.** In this example, we show that the state equivalence relation $\equiv_e$ corresponds to our desired $\equiv$ relation. We refer to the four axioms given in Definition 8.2 via $\equiv_1, \equiv_2, \equiv_3, \text{ and } \equiv_4$.

- *Equivalence III.1:* $\langle c(X); \top; \emptyset \rangle \equiv_2 \langle c(X); X = Y; \emptyset \rangle \equiv_1 \langle c(Y); X = Y; \emptyset \rangle \equiv_2 \langle c(Y); \top; \emptyset \rangle$
- *Equivalence III.2* follows directly from $\equiv_1$.
- *Equivalence III.3* follows directly from $\equiv_2$.
- *Equivalence III.4* follows directly from $\equiv_3$.
- *Equivalence III.5:* Using an axiomatic definition for $\equiv_e$ makes it difficult to prove non-equivalence. While an equivalence proof requires a sequence of axiom applications, proving non-equivalence requires us to show that no such sequence exists. Therefore, we delay the proof that $\equiv_e$ satisfies the non-equivalence III.5, until we have the decision criterion, developed in the following section, available.

### 8.1.3 Deciding State Equivalence

The decision problem for state equivalence is: given two states $\sigma_1$ and $\sigma_2$ decide whether $\sigma_1 \equiv_e \sigma_2$. In the positive case of two equivalent states, this can be decided by finding suitable applications of the axioms of Definition 8.2 that transform one state into the other. However, given two non-equivalent states, an axiom-based proof would require showing that no such applications can exist. As this is hard to automate, we would like to have a better criterion for a decision algorithm.

Logical equivalence between $\exists \bar{y}. G \land B$ and $\exists \bar{y}'. G' \land B'$ is a necessary but not a sufficient condition for state equivalence between $\langle G; B; V \rangle$ and $\langle G'; B'; V' \rangle$ (cf. Lemma 8.3:3). This is due to the fact that unlike logical equivalence, state equivalence preserves the multiplicities of logically equivalent user-defined constraints. A similar condition, which is also sufficient can be formulated in *linear-logic* [Betz and Frühwirth, 2005].

Theorem 3 gives a necessary and sufficient criterion for deciding state equivalence. Due to its preconditions, it technically decides a smaller relation than $\equiv_e$, because it only applies to the case that local variables are renamed apart and the set of global variables is unchanged. However, this restriction is not problematic for deciding equivalence in general. By Lemma 8.3 we are free to rename local variables apart and by Def. 8.2:3 we can adjust the sets of global variables to match. Therefore, Theorem 3 gives us a necessary and sufficient criterion for equivalence of arbitrary states: first we transform the states into equivalent states that satisfy the preconditions, then we apply the theorem. The transformation is straightforward and equivalence-preserving, hence, the result we get from the theorem applies to the original states by transitivity of $\equiv_e$. Finally, decidability of our criterion is a direct consequence of decidability of $\mathcal{CT}$. 

38
**Theorem 3** (Criterion for $\equiv_e$). Let $\sigma = \langle G; B; V \rangle$, $\sigma' = \langle G'; B'; V \rangle$ be $\omega_e$ states with local variables $\bar{y}, \bar{y}'$ that have been renamed apart.

$$\sigma \equiv_e \sigma'$$

if and only if

$$CT \models \forall(B \to \exists \bar{y}'.((G = G') \land B')) \land \forall(B' \to \exists \bar{y}.((G = G') \land B))$$

**Proof.** Let $C$ be a binary predicate on CHR states such that $C(\langle G; B; V \rangle, \langle G'; B'; V \rangle)$ holds iff

$$CT \models \forall(B \to \exists \bar{y}'.((G = G') \land B')) \land \forall(B' \to \exists \bar{y}.((G = G') \land B))$$

We show that each of the three implicit conditions – reflexivity, symmetry, and transitivity – as well as the four explicit conditions of Def. 8.2 are sound w.r.t. criterion $C$.

**Reflexivity:** Reflexivity is given as the following judgment is clearly true:

$$CT \models \forall(B \to \exists \bar{y}.((G = G) \land B)) \land \forall(B \to \exists \bar{y}.((G = G) \land B))$$

**Symmetry:** Symmetry of $C$ is obvious.

**Transitivity:** Assume three states $\sigma = \langle G; B; V \rangle$, $\sigma' = \langle G'; B'; V \rangle$, $\sigma'' = \langle G''; B''; V \rangle$ with distinct local variables $\bar{y}, \bar{y}', \bar{y}''$ such that $C(\sigma, \sigma')$ and $C(\sigma', \sigma'')$. By definition, we have:

$$CT \models \forall(B \to \exists \bar{y}'.((G = G') \land B')) \quad (i)$$

$$CT \models \forall(B' \to \exists \bar{y}.((G = G') \land B)) \quad (ii)$$

$$CT \models \forall(B' \to \exists \bar{y}'.((G' = G'') \land B')) \quad (iii)$$

$$CT \models \forall(B'' \to \exists \bar{y}'.((G' = G'') \land B')) \quad (iv)$$

From (i) and (iii) follows:

$$CT \models \forall(B \to \exists \bar{y}''.((G = G'') \land B''))$$

From (ii) and (iv) follows:

$$CT \models \forall(B'' \to \exists \bar{y}.((G = G'') \land B))$$

Consequently, $C(\sigma, \sigma'')$ holds.

**Equality as Substitution:** Assume two states $\sigma = \langle G; X = t \land B; V \rangle$, $\sigma' = \langle G[X/t]; X = t \land B; V \rangle$ with local variables $\bar{y}, \bar{y}'$. As $CT \models \forall(X = t \to (G = G[X/t]))$, we have $C(\sigma, \sigma')$.

**Transformation of the Constraint Store:** Assume two states $\sigma = \langle G; B; V \rangle$ and $\sigma' = \langle G; B'; V \rangle$ with local variables $\bar{y}, \bar{y}'$ and strictly local variables $\bar{s}, \bar{s}'$ such that $CT \models \exists \bar{s}.B \leftrightarrow \exists \bar{s}'.B'$. This implies the following judgment:

$$CT \models \forall(B \to \exists \bar{y}'.((G = G) \land B')) \land \forall(B' \to \exists \bar{y}.((G = G) \land B))$$

Hence, $C(\sigma, \sigma')$ holds.

**Omission of Non-Occurring Global Variables:** Does not apply since $\sigma$ and $\sigma'$ share the set $V$ of global variables.
Equivalence of Failed States: For any two failed states, they are of the form $\langle G; \bot; V \rangle$ and $\langle G'; \bot; V' \rangle$. The following judgment proves $C((\langle G; \bot; V \rangle, (\langle G'; \bot; V' \rangle))$:

$$CT \models \forall (\bot \rightarrow \exists \bar{y}'.((G = G') \land \bot)) \land \forall (\bot \rightarrow \exists \bar{y} .((G = G') \land \bot))$$

$\Leftarrow$:

We consider two CHR states $\sigma = \langle G; B; V \rangle, \sigma' = \langle G'; B'; V \rangle$ with disjunct local variables $\bar{y}$ and $\bar{y}'$. We assume that

$$CT \models \forall (B \rightarrow \exists \bar{y}'.((G = G') \land B')) \land \forall (B' \rightarrow \exists \bar{y} .((G = G') \land B))$$

If there does not exist a pairwise matching $G = G'$, we have $B = B' = \bot$, which proves that $\sigma \equiv_{e} \sigma'$ by Def. 8.2:4. In the following, we assume that a pairwise matching $G = G'$ does exist.

It follows from $\forall (B \rightarrow \exists \bar{y}'.((G = G') \land B'))$ by Def. 8.2:2 that:

$$\sigma \equiv_{e} \langle G; G' \land B \land B'; V \rangle$$

By Def. 8.2:1 we have:

$$\sigma \equiv_{e} \langle G'; G = G' \land B \land B'; V \rangle$$

From $\forall (B' \rightarrow \exists \bar{y} .((G = G') \land B))$ we get by Def. 8.2:2 that:

$$\sigma \equiv_{e} \langle G'; B'; V \rangle = \sigma'$$

Example 8.5. We can now use Theorem 3 to prove our desired non-equivalence (III.5). Let $\sigma = \langle c(X); \top; \{X\} \rangle$ and $\sigma' = \langle c(Y); \top; \{Y\} \rangle$, then we first consider two states with the same global variables as follows:

$$\sigma_1 := \langle c(X); \top; \{X, Y\} \rangle \equiv_{e} \sigma$$

$$\sigma_2 := \langle c(Y); \top; \{X, Y\} \rangle \equiv_{e} \sigma'$$

According to the above theorem, $\sigma_1 \equiv_{e} \sigma_2$ holds if and only if

$$CT \models \forall X, Y. \top \rightarrow (c(X) = c(Y)) \land \forall X, Y. \top \rightarrow (c(X) = c(Y)).$$

However, it clearly holds that $CT \not\models \forall X, Y. c(X) = c(Y)$, such that Theorem 3 proves $\sigma_1 \not\equiv_{e} \sigma_2$, and because $\equiv_{e}$ is an equivalence relation, this proves $\sigma \not\equiv_{e} \sigma'$.

Intuitively, the theorem tries to find out if the two multisets of CHR constraints can be made syntactically equivalent ($G = G'$). However, we may only consider existential quantification for local variables, which subsumes variable renaming and substitution. Interpreting $G = G'$ as a unification, we must additionally ensure that the required bindings satisfy the other state’s built-in store.

Theorem 3 is indeed suitable for implementation. Such an implementation has been given in [Langbein et al., 2010], where it was used as the foundation of a confluence checker.
Chapter III

Excursion: Correction of Book
In [Frühwirth, 2009, p.71] a similar criterion is given. However, it is not used as a decision criterion, but instead as a definition of state equivalence. By that definition two states \( \sigma = \langle G; B; V \rangle \) and \( \sigma' = \langle G'; B'; V \rangle \) with local variables \( \bar{y}, \bar{y}' \) that have been renamed apart are equivalent if and only if
\[
\mathcal{CT} \models \forall (B \rightarrow \exists \bar{y}. (G = G')) \land \forall (B' \rightarrow \exists \bar{y}'. (G = G')) \land \forall (\exists \bar{y}. B \leftrightarrow \exists \bar{y}'. B')
\]
However, this definition results in a different relation with non-intuitive implications. Consider for example the states \( \sigma = \langle c(X); X = 3; \emptyset \rangle \) and \( \sigma' = \langle c(Y); Y = 2; \emptyset \rangle \). Intuitively, these states should not be equivalent, however, it holds that
- \( \mathcal{CT} \models \forall X. X = 3 \rightarrow \exists Y. (c(X) = c(Y)) \)
- \( \mathcal{CT} \models \forall Y. Y = 2 \rightarrow \exists X. (c(X) = c(Y)) \)
- \( \mathcal{CT} \models (\exists X. X = 3) \leftrightarrow (\exists Y. Y = 2) \)
Therefore, all conditions are satisfied such that \( \sigma \) and \( \sigma' \) are equivalent according to that definition. Clearly, this has not been intended and one should instead rely on Theorem 3.

8.2 State Transition System
In this section, we first define an operational semantics for CHR in Section 8.2.1 based on our axiomatic definition of state equivalence. Section 8.2.2 then discusses its equivalence to the traditional definition, hence showing that state equivalence is indeed compliant with rule application. Finally, we introduce a novel view of the CHR transition system in Section 8.2.3: Given the compliance of state equivalence with rule application, we can integrate this knowledge into the formulation of the operational semantics, hence, interpreting CHR as a rewriting system of equivalence classes.

8.2.1 Equivalence-based Transition System
In this section, we present a formulation of the operational semantics based on state equivalence. Our definition is not only based on the traditional definition, but is also provably equivalent. Integrating the notion of state equivalence permits removing the matching that has traditionally been hidden in the complex formula \( H_1 = H'_1 \land H_2 = H'_2 \). Furthermore, imposing a guard condition on \( \mathcal{CT} \) becomes dispensable, leading to the following simplified operational semantics:

**Definition 8.6** (Operational Semantics \( \omega_e \)). For a CHR program \( \mathcal{P} \) we define the state transition system \( (\Sigma_e, \rightarrow_e) \), referred to as \( \omega_e \), as given in Table III.2. The transition is based on a variant of a rule \( r \) in \( \mathcal{P} \) such that its local variables are disjoint from the variables occurring in the pre-transition state.

When the rule \( r \) is clear from the context or not important, we may write \( \rightarrow_e r \) rather than \( \rightarrow_e r_e \). By \( \rightarrow_e^* \), we denote the reflexive-transitive closure of \( \rightarrow_e \).
8. Equivalence-based Operational Semantics

\[
\begin{align*}
H_1 \setminus H_2 & \leftrightarrow G \mid B_c, B_b \\
(H_1 \cup H_2 \cup G; G \land B; V) & \rightarrow_e (H_1 \cup B_c \cup G; G \land B_b; V) \\
\tau' & \equiv \tau
\end{align*}
\]

Table III.2: State Transition System $\omega_e$

Example 8.7. Consider again the example for computing the greatest common divisor from Example 5.6, which consists of the following two rules.

\[
\begin{align*}
gcd_1 & @ gcd(0) \leftrightarrow \top \\
gcd_2 & @ gcd(N) \setminus gcd(M) \leftrightarrow M \geq N \land N > 0 \mid gcd(L), L = M \% N
\end{align*}
\]

Given the transition system $\rightarrow_e$, we can now derive the greatest common divisor of 6 and 15 as follows:

\[
\begin{align*}
\langle gcd(6), gcd(15); \top; \emptyset \rangle \\
\langle gcd(N), gcd(M); M \geq N \land N > 0 \land N = 6 \land M = 15; \emptyset \rangle \\
\langle gcd(6), gcd(3); \top; \emptyset \rangle \\
\langle gcd(N), gcd(M); M \geq N \land N > 0 \land L = M \% N \land N = 6 \land M = 15; \emptyset \rangle \\
\langle gcd(6), gcd(3); \top; \emptyset \rangle \\
\langle gcd(N), gcd(M); M \geq N \land N > 0 \land L = M \% N \land N = 3 \land M = 6; \emptyset \rangle \\
\langle gcd(3), gcd(0); \top; \emptyset \rangle \\
\langle gcd(3); \top; \emptyset \rangle
\end{align*}
\]

We can see from the above derivation that three rules have been applied. We reuse the variables $N, M$ in the derivation above, because they can be removed from the state after the rule application, by applying the equivalence relation.

The most significant difference of this formulation of the operational semantics to previous ones is that the rule head is required to exist within the state exactly as is. Traditionally, we require constraints in the state that are similar to the ones in the rule head. They are then matched via a substitution. With Definition 8.6 however, this is outsourced into the equivalence relation. We still have to prove that the current state contains the required rule head, but the formulation of the operational semantics itself is simplified.

Another advantage can be seen in the previous example: during a derivation we can use the equivalence relation to freely switch to different representations of a state. This, for example, allows us to constantly simplify the built-in store. When we compare this to $\omega_{va}, \omega_t$, and $\omega_p$, we find that in $\omega_{va}$ there is no formal means to ever simplify a built-in store again, while $\omega_t$ and $\omega_p$ may only simplify the built-in store during a Solve transition.

Example 8.8. To demonstrate the tremendous simplification gained by $\omega_e$ let us consider the previous derivation for computing the greatest common divisor of 6 and 15 for $\omega_t$. For space reason, we denote $L_1 = 3 \land N_2 = L_1 \land M_2 = 6 \land M_2 \geq N_2 \land N_2 > 0$ as $B$ in the below
derivation.

\[
\begin{align*}
\text{Introduce} & \quad (\gcd(6), \gcd(15); \emptyset; \top; \emptyset)^0_i \nonumber \\
\text{Introduce} & \quad (\gcd(15); \gcd(6); \emptyset; \top; \emptyset)^0_i \nonumber \\
\text{Introduce} & \quad (\emptyset; \gcd(6) \# 0, \gcd(15) \# 1; \top; \emptyset)^0_i \nonumber \\
\text{Solve} & \quad (\gcd(L_1); \gcd(6) \# 0; L_1 = 3; (\gcd_{d_2}, [0, 1]));^0_i \\
\text{Introduce} & \quad (\emptyset; \gcd(6) \# 0, \gcd(15) \# 2; L_1 = 3; (\gcd_{d_2}, [0, 1]));^0_i \\
\text{Solve} & \quad (\gcd(L_1); \gcd(6) \# 0; L_1 = 2; (\gcd_{d_2}, [0, 1]));^0_i \\
\text{Introduce} & \quad (\emptyset; \gcd(6) \# 0, \gcd(L_1) \# 2; L_1 = 3; (\gcd_{d_2}, [0, 1]));^0_i \\
\text{Solve} & \quad (\gcd(L_1); \gcd(6) \# 0; L_1 = 2; (\gcd_{d_2}, [0, 1]));^0_i \\
\text{Solve} & \quad (\emptyset; \gcd_{d_2}(L_1) \# 2; L_1 = 3 \land L_2 = 0; (\gcd_{d_2}, [0, 1]));^0_i \\
\vdots & \\
\vdots & \\
\vdots & \\
\end{align*}
\]

Apart from the higher complexity of this derivation, there is an important point about the resulting final state: For the above, one often reads abbreviated derivations like

\[
(g \text{cd}(6), g \text{cd}(15); \emptyset; \top; \emptyset)^0_i \rightarrow (\emptyset; g \text{cd}(3) \# i; \emptyset; \top)^0_i,
\]

for some \(i, \emptyset \in \mathbb{T}, \top, \) and \(n\). However, the above statement is in fact wrong, because there exist no such derivations according to Definition 5.9. Instead there only exists a constraint \(g \text{cd}(L_1) \# 3\) and the built-in \(L_1 = 3\). With \(\omega_i\) we are not given any means to change this into a constraint of the form \(g \text{cd}(3)\).

The same problem exists with \(\omega_{\text{va}}\), where we have seen in Example 5.6 that we also cannot get a \(g \text{cd}(3)\) constraint in the resulting final state. In fact, the situation is worse for \(\omega_{\text{va}}\), as there is no built-in simplification possible at all, whereas \(\omega_i\) at least has the \textbf{Solve} transition available.

### 8.2.2 Compliance to Rule Application

Definition 8.6 implicitly assumes that when we can apply a rule to a state, we can also apply it to all equivalent states. Furthermore, the resulting states of the rule application are then assumed to be equivalent as well. This is by no means a trivial assumption. While it is a property that we would intuitively expect from a proper equivalence relation, it has not been proven before the introduction of \(\omega_e\) in \cite{Raiser et al., 2009}.

The following theorem from \cite{Raiser et al., 2009} proves the equivalence of \(\omega_e\) and \(\omega_{\text{va}}\). This implicitly proves the above desired property. Hence, it effectively serves as justification for the viability of Definition 8.6.

**Theorem 4 (Equivalence of the Definitions).** For a CHR state \(\sigma\) we have

1. If \(\sigma \rightarrow^* e \tau\) then there exists a state \(\tau' \equiv \tau\) with \(\sigma \rightarrow^*_{\text{va}} \tau'\)
2. If \(\sigma \rightarrow^*_{\text{va}} \tau'\) then there exists a state \(\tau \equiv \tau'\) with \(\sigma \rightarrow^* e \tau\)

**Proof.** see \cite{Raiser et al., 2009}.

### 8.2.3 Rewriting of Equivalence Classes

By now, we have introduced an axiomatic definition of state equivalence and shown its compliance with rule applications. Definition 8.6 of \(\omega_e\) can therefore be abstracted further. The second inference rule allows us to freely switch between equivalent states during a derivation. Therefore, the actual syntactical representation of a state is of no importance anymore,
Recent work on linear-logical algorithms [Simmons and Pfenning, 2008] and the close relation of CHR to linear-logic [Betz and Frühwirth, 2005] suggest a novel approach that emphasizes aspects from both sides of the spectrum to a useful degree: In [Betz et al., 2009], we introduce the notion of persistent constraints to CHR, a concept reminiscent of unrestricted or “banged” propositions in linear-logic. Persistent constraints provide a finite representation of the result of any number of propagation rule firings.

We furthermore introduce a state transition system based on persistent constraints, which is explicitly irreflexive. In combination, the two ideas solve the problem of trivial non-termination while retaining declarativity and preserving the potential for effective concurrent
execution. This state transition system requires no more than two rules. As every transition step corresponds to a CHR rule application, it facilitates formal reasoning over programs. In this work, we show that the resulting operational semantics $\omega_1$ is sound and complete with respect to $\omega_e$. We show that $\omega_1$ can be faithfully embedded into the operational semantics $\omega_p$, thus effectively providing an implementation in the form of a source-to-source transformation. All operational semantics developed with an emphasis on pragmatic aspects lack this completeness property. Therefore, it is possible to implement CHR soundly and completely with respect to its abstract foundations, whilst featuring a terminating execution model for propagation rules.

Example 9.1. Consider the following straightforward CHR program for computing the transitive hull of a graph represented by edge constraints $e/2$:

$$t @ e(X,Y), e(Y,Z) \implies e(X,Z)$$

This most intuitive formulation of a transitive hull is not a suitable implementation in most existing operational semantics. In fact, for goals containing cyclic graphs it is non-terminating in all aforementioned existing semantics. In this work we show that execution in our proposed semantics $\omega_1$ correctly computes the transitive hull whilst guaranteeing termination.

9.1 State Equivalence

In this section, we present the operational semantics $\omega_1$ with persistent constraints, originally proposed in [Betz et al., 2009]. It is based on the following ideas:

1. In $\omega_e$, the body of a propagation rule can be generated any number of times, provided that the corresponding head constraints are present in the store. In order to give consideration to this theoretical behavior, we introduce those body constraints as so-called persistent constraints. A persistent constraint is a finite representation of a large, though unspecified number of identical constraints. For a proper distinction, constraints that are not persistent constraints are henceforth called linear constraints.

2. As a secondary consequence, arbitrary generation of rule bodies in $\omega_e$ affects other types of CHR rules as well. Consider the following program:

$$r1 @ a \implies b$$
$$r2 @ b \iff c$$

If executed with goal $a$, this program can generate an arbitrary number of $b$-constraints. As a consequence of this, it can also generate arbitrarily many $c$-constraints. To take these indirect consequences of propagation rules into account, we introduce a rule’s body constraints as persistent, whenever its removed head can be matched completely with persistent constraints.

3. As a persistent constraint represents an arbitrary number of identical constraints, we consider multiple occurrences of a persistent constraint as idempotent. Thus, we implicitly apply a set semantics to persistent constraints.

4. We adapt the execution model such that a transition takes place only if the post-transition state is not equivalent to the pre-transition state. This entails two beneficial
consequences: Firstly, in combination with the set semantics on persistent constraints, it avoids trivial non-termination of propagation rules. Secondly, as failed states are equivalent, it enforces termination upon failure.

We adapt the definition of \( \omega \) states with respect to \( \omega_e \). The goal store \( G \) of \( \omega_e \) states is split into a store \( L \) of linear constraints and a store \( P \) of persistent constraints.

**Definition 9.2 (\( \omega \) State).** A \( \omega \) state is a tuple of the form \( \langle L; P; B; V \rangle \), where \( L \) and \( P \) are multisets of CHR constraints called the linear (CHR) store and persistent (CHR) store, respectively. \( B \) is a conjunction of built-in constraints and \( V \) is a set of variables called the local variables. We use \( \Sigma \) to denote the set of all \( \omega \) states.

The following definition of state equivalence is adapted to comply with Definition 9.2 and to handle idempotence of persistent constraints.

**Definition 9.3 (Equivalence of \( \omega \) States).** Equivalence between \( \omega \) states is the smallest equivalence relation \( \equiv \) over \( \omega \) states that satisfies the following conditions:

1. (Equality as Substitution)
   \[ \langle L; P; X = t \land B; V \rangle \equiv \langle L[X/t]; P[X/t]; X = t \land B; V \rangle \]

2. (Transformation of the Constraint Store) If \( CT \models \exists \bar{s}, B \leftrightarrow \exists \bar{s}', B' \) where \( \bar{s}, \bar{s}' \) are the strictly local variables of \( B, B' \), respectively, then:
   \[ \langle L; P; B; V \rangle \equiv \langle L; P; B'; V \rangle \]

3. (Omission of Non-Occurring Global Variables) If \( X \) is a variable that does not occur in \( L, P, \) or \( B \) then:
   \[ \langle L; P; B; \{ X \} \cup V \rangle \equiv \langle L; P; B; V \rangle \]

4. (Equivalence of Failed States)
   \[ \langle L; P; \bot; V \rangle \equiv \langle L'; P'; \bot; V' \rangle \]

5. (Contraction)
   \[ \langle L; P \uplus P \uplus P; B; V \rangle \equiv \langle L; P \uplus P; B; V \rangle \]

The following definition presents an auxiliary concept that we use to formulate a criterion for \( \omega \) equivalence, in analogy to Theorem 3. Intuitively, \( G \bowtie G' \) holds, if after converting \( G \) and \( G' \) to sets by eliminating duplicates, the two sets are identical.

**Definition 9.4 (\( \bowtie \)).** The relation \( \bowtie \) over multisets of constraints is defined as

\[ G \bowtie G' \text{ if and only if } (\forall c \in G. \exists c' \in G'. c = c') \land (\forall c' \in G'. \exists c \in G. c = c') \]

**Theorem 5 (Criterion for \( \equiv \)).** Let \( \sigma = \langle L; P; B; V \rangle \), \( \sigma' = \langle L'; P'; B'; V \rangle \) be \( \omega \) states with local variables \( \bar{y}, \bar{y}' \) that have been renamed apart.

\[ \sigma \equiv \sigma' \]

if and only if

\[ CT \models (B \rightarrow \exists \bar{y}'.((L = L') \land (P \bowtie P') \land B')) \land (B' \rightarrow \exists \bar{y}.((L = L') \land (P \bowtie P') \land B)) \]
Proof.

'⇐': We consider two \( \omega_1 \) states \( \sigma = \langle L; P; B; V \rangle, \sigma' = \langle L'; P'; B'; V \rangle \) with local variables \( \bar{y} \) and \( \bar{y}' \). We furthermore assume that:

\[
CT \models \forall (B \to \exists \bar{y}'. ((L = L') \land (P \bowtie P') \land B')) \land \forall (B' \to \exists \bar{y}. ((L = L') \land (P \bowtie P') \land B))
\]

If \( CT \models \neg \exists ((L = L') \land (P \bowtie P')) \), we have \( CT \models B = B' = \bot \) such that Def. 9.3:4 proves \( \sigma \equiv \sigma' \). In the following, we assume that a matching \( (L = L') \land (P \bowtie P') \) exists.

It follows from \( \forall (B \to \exists \bar{y}'. ((L = L') \land (P \bowtie P') \land B')) \) by Def. 9.3:2 that:

\[
\sigma \equiv \langle L; P; (L = L') \land (P \bowtie P') \land B \land B'; V \rangle
\]

Def. 9.3.1 gives us:

\[
\sigma \equiv \langle L'; P'; (L = L') \land (P \bowtie P') \land B \land B'; V \rangle
\]

where \( P'' \) equals \( P' \) modulo multiplicities. By Def. 9.3:5 we thus get:

\[
\sigma \equiv \langle L'; P'; (L = L') \land (P \bowtie P') \land B \land B'; V \rangle
\]

From \( \forall (B' \to \exists \bar{y}'. ((L = L') \land (P \bowtie P') \land B)) \) follows by Def. 9.3:2 that:

\[
\sigma \equiv \langle L'; P'; B'; V \rangle = \sigma'
\]

'⇒': To prove the forward direction, we have to show the compliance of the conditions in Def. 9.3.1 to Def. 9.3:5 with our criterion. For Def. 9.3:1 to Def. 9.3:4, compliance is analogous to Thm. 3. Hence, we now consider Def. 9.3:5:

Let \( \sigma = \langle L; P \uplus P \uplus P; B; V \rangle, \sigma' = \langle L; P \uplus P; B; V \rangle \in \Sigma_1 \) with local variables \( \bar{y}, \bar{y}' \). As, \( (P \uplus P \uplus P) \bowtie (P \uplus P) \bowtie (P \uplus P) \bowtie (P \uplus P) \bowtie B \), the following is a tautology:

\[
CT \models \forall (B \to \exists \bar{y}'. ((L = L) \land ((P \uplus P \uplus P) \bowtie (P \uplus P) \bowtie B))) \land
\forall (B \to \exists \bar{y}. ((L = L) \land ((P \uplus P \uplus P) \bowtie (P \uplus P) \bowtie B)))
\]

\( \square \)

9.2 State Transition System

Based on the definition of \( \omega_1 \), we define the operational semantics \( \omega_1 \) below. Since body constraints may be introduced either as linear or as persistent constraints, uniform rule application is replaced by two distinct application modes. An important restriction is that \( \omega_1 \) is only defined for range-restricted programs (cf. Section 11.2 for details).

**Definition 9.5 (\( \omega_1 \) Transitions).** For a range-restricted CHR program \( P \), the state transition system \( (\Sigma_1; \equiv_1; \rightarrow_1) \), referred to as \( \omega_1 \), is given in Table III.4.

When the rule \( r \) is clear from the context or not important, we may write \( \rightarrow_1 \) rather than \( \rightarrow_1^r \). By \( \rightarrow_1^\ast \), we denote the reflexive-transitive closure of \( \rightarrow_1 \).
9. Constraint Handling Rules with Persistent Constraints

ApplyLinear:

\[
r \otimes (H^1_a \cup H^p_1) \backslash (H^2_a \cup H^p_2) \iff G \mid B_c, B_b \quad H_1^a \not= \emptyset \quad [\sigma] \neq [\tau]
\]

\[
[\sigma] = ([H^1_a \cup H^p_1 \cup \emptyset; H^2_b \cup H^p_2 \cup \emptyset; G \land B \land V])
\]

\[
\Rightarrow \Rightarrow ([H^1_a \cup B_c \cup L; H^2_b \cup H^p_2 \cup \emptyset; G \land B \land B_b; V]) = [\tau]
\]

ApplyPersistent:

\[
r \otimes (H^1_a \cup H^p_1) \backslash (H^2_a \cup H^p_2) \iff G \mid B_c, B_b \quad [\sigma] \neq [\tau]
\]

\[
[\sigma] = ([H^1_a \cup \emptyset; H^p_1 \cup H^p_2 \cup \emptyset; G \land B \land V])
\]

\[
\Rightarrow \Rightarrow ([H^1_a \cup \emptyset; H^p_1 \cup H^p_2 \cup B_c \cup \emptyset; G \land B \land B_b; V]) = [\tau]
\]

Table III.4: State Transition System \(\omega_1\)

Example 9.6. Again consider the transitive edge program from Example 9.1 and an initial state that contains a cyclic graph. For such an initial state the program is non-terminating for all previous operational semantics presented in this work. However, with \(\omega_1\), we get the following terminating derivation.

\[
[\sigma] = [(e(A, B), e(B, A) ; \emptyset ; \emptyset ; \{A, B\})]
\]

\[
\Rightarrow [(e(A, B), e(B, A) ; e(A, A) ; \emptyset ; \{A, B\})]
\]

\[
\Rightarrow [(e(A, B), e(B, A) ; e(A, A) , e(B, B) ; \emptyset ; \{A, B\})]
\]

\[
\Rightarrow [(e(A, B), e(B, A) ; e(A, A) , e(B, B) , e(A, B) ; \emptyset ; \{A, B\})]
\]

\[
\Rightarrow [(e(A, B), e(B, A) \setminus e(A, A) , e(B, B) , e(A, B) , e(B, A) ; \emptyset ; \{A, B\})] = [\tau]
\]

All four state transitions are made via the ApplyPersistent inference rule and add the resulting constraints to the persistent store. Irreflexivity of \((\Sigma_1/\equiv_1, \Rightarrow_1)\) ensures that such a persistent constraint cannot be generated again. Therefore, after all four possible edges have been generated as persistent constraints, no further rule applications are possible.

This operational semantics \(\omega_1\) fills a gap left by the existing operational semantics: It is, on the one hand, complete, i.e. every computation that is possible under the operational semantics \(\omega_c\) corresponds to one in \(\omega_1\). On the other hand, it offers general termination for propagation rules, which \(\omega_c\) lacks and caused the token-based operational semantics \(\omega_t\) to be developed, which in turn lacks completeness.

This situation is easily demonstrated via a propagation rule \(a \Rightarrow b\) and an initial state that contains an \(a\)-constraint. With \(\omega_c\) it is possible to apply the rule any number of times. In particular, we get a non-terminating derivation which keeps generating \(b\)-constraints. In the more pragmatic operational semantics \(\omega_t\) instead, we can only derive a single \(b\)-constraint. While we have termination in that case, we lack completeness in that we are unable to generate a second or third \(b\)-constraint.

The operational semantics \(\omega_1\), however, fixes these problems with a best-of-both-worlds approach: in a single derivation step a persistent \(b\)-constraint is derived, hence, we have termination. Furthermore, this persistent constraint can be used in lieu of any number of linear \(b\)-constraints. If we add to the above rule, the rule \(a, b \Leftrightarrow c\), then \(\omega_1\) allows a second derivation that yields a linear \(c\)-constraint. Again, \(\omega_1\) fails due to its lack of completeness to ever derive a \(c\)-constraint, and \(\omega_c\) still allows the non-terminating derivation, but also terminating derivations that yield a \(c\)-constraint.
Of course, $\omega_f$ is also sound with respect to $\omega_e$, i.e. all derivations made in $\omega_f$ correspond to derivations in $\omega_e$. For a formal treatment of the soundness and completeness of $\omega_f$ with respect to $\omega_e$ we refer the reader to [Betz et al., 2010b].

**Excursion: Proverbial Example**

This excursion demonstrates the potential of persistent constraints in a less serious way. Consider the following well-known Chinese proverb, which we want to model as a CHR program.

Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.

First of all, there are implicit facts in this proverb, namely, that there can be hungry men and that eating a fish saturates them. This is straightforwardly expressed in CHR as

$$r_1 @ \text{hungry}(M), \text{fish} \iff \text{saturated}(M).$$

Assuming days roughly correspond to rule applications, the above rule nicely captures the idea that you can give a fish to a man $M$ in order to feed him. Of course, he eventually gets hungry again:

$$r_2 @ \text{saturated}(M) \iff \text{hungry}(M).$$

Now, we can consider the input $\langle \text{hungry}(M), \text{fish}; \emptyset; \top; \{M\} \rangle$, and unsurprisingly, this is a case of starvation – on the one hand, because we can only apply each of the two rules once, on the other hand, because the poor man has no more fish to eat.

Next, let us model what happens, if a man is taught to fish. We assume that the man is saturated, when we teach him to fish, and the result is clear: The man becomes a fisher.

$$r_3 @ \text{saturated}(M) \setminus \text{teach}(M) \iff \text{fisher}(M).$$

Of course, any fisher can follow his profession and go fishing. At this point, we make the forgivable assumption that our fisher is as extraordinary as the supply of fish, such that he can catch a fish whenever he wants to.

$$r_4 @ \text{fisher}(M) \implies \text{fish}$$

We have now successfully modeled the second sentence of the above proverb: Given a man and someone willing to teach him to fish ($\langle \text{saturated}(M), \text{teach}(M); \emptyset; \top; \{M\} \rangle$), that man can feed himself for a lifetime (non-terminating computation).

Our model is quite realistic, as it also accounts for other phenomenons:

- What if the man refuses to be taught? By non-determinism and rule $r_2$, the man will become hungry, and at this point he has a serious starvation problem.

- What happens if one tries to teach a hungry man? As none of the rules is applicable in that case, he will simply starve. So we should at least bring one fish along to be safe.

Finally, when we consider several hungry men, we realize that our model solves the problem of nourishing the world’s population: All we need is a single fisher – and if he is fair, he will even feed the immortals.
10 Merge Operator

In this section, we present the merge operator $\triangleright$ that combines two CHR states. It is a beneficial tool in formal program analysis and will find application throughout Chapter IV. Here, we analyze its properties in Section 10.1, before we discuss its implied partial order on states in Section 10.2. All results given in this section are formulated for $\omega$, but work equally for $\omega_\epsilon$, when simply considering a projection of states that ignores the persistent store.

**Definition 10.1** (Merge Operator $\triangleright$). Let $\sigma_1 = (L_1; P_1; B_1; V_1)$ and $\sigma_2 = (L_2; P_2; B_2; V_2)$ such that local variables of one state are disjunct from all variables in the other state. Then for a set $V$ of variables

$$\sigma_1 \triangleright V \sigma_2 := (L_1 \uplus L_2; P_1 \uplus P_2; B_1 \land B_2; (V_1 \cup V_2) \setminus V).$$

We further lift this definition to equivalence classes. In that case, the merge operation assumes that two representants with accordingly disjunct variables are selected:

$$[\sigma_1] \triangleright V [\sigma_2] := [\sigma_1 \triangleright V \sigma_2].$$

For $V = \emptyset$, we write $\sigma_1 \triangleright \sigma_2$ and $[\sigma_1] \triangleright [\sigma_2]$, respectively.

Applying the merge operator on equivalence classes assumes that two representants are selected that satisfy the above condition for their variables. Lemma 8.3 shows that renaming of local variables keeps equivalence of states, hence, such representants are guaranteed to exist.

**Example 10.2.** Merging $(c(X); 0; T; 0)$ and $(0; 0; X = 1; 0)$ should result in $(c(X); 0; X = 1; 0)$. However, when considering equivalence classes we would instead get $[(c(X); 0; T; 0)]$, because $(0; 0; X = 1; 0) \equiv_1 (0; 0; T; 0)$.

For that reason, the above definition restricts local variables to one state and allows turning global variables into local variables during the merge operation. Hence, we can perform the following merge operation:

$$[(c(X); 0; T; \{X\})] \triangleright X [(0; 0; X = 1; \{X\})] = [(c(X); 0; X = 1; 0)]$$

10.1 Properties of the Merge Operator

Lifting the merge operator $\triangleright$ to equivalence classes is justified by the following lemma, which shows that it maintains state equivalence.

**Lemma 10.3** ($\triangleright V$ maintains Equivalence). Let $\sigma_1 \equiv_1 \sigma_2$, then $(\sigma_1 \triangleright V \tau) \equiv_1 (\sigma_2 \triangleright V \tau)$ for all $V$.

**Proof.** W.l.o.g. let $\sigma_i = (L_i; P_i; B_i; V')$ for $i = 1, 2$ and let $\tau = (L; P; B; \bar{V})$ such that the variables are disjunct according to Def. 10.1. Let $\bar{y}_1, \bar{y}_2$ be the local variables of $\sigma_1$ and $\sigma_2$ respectively. We know by Thm. 5 that:

$$CT \models \forall (B_1 \rightarrow \exists \bar{y}_2.((L_1 = L_2) \land (P_1 \triangleright \exists P_2) \land B_2)) \land$$

$$\forall (B_2 \rightarrow \exists \bar{y}_1.((L_1 = L_2) \land (P_1 \triangleright \exists P_2) \land B_1))$$

Let $\bar{x} = (V' \cap V)$, then

$$CT \models \forall (B_1 \rightarrow \exists \bar{y}_2 \exists \bar{x}.((L_1 = L_2) \land (P_1 \triangleright \exists P_2) \land B_2)) \land$$

$$\forall (B_2 \rightarrow \exists \bar{y}_1 \exists \bar{x}.((L_1 = L_2) \land (P_1 \triangleright \exists P_2) \land B_1))$$
As \((L = \mathbb{L})\) and \((\mathbb{P} \bowtie \mathbb{P})\) are tautologies, we can extend \((L_1 = L_2)\) to \((\mathbb{L} \triangleleft L) = (\mathbb{L} \triangleright L)\)
and analogously for \(\mathbb{P}\). Similarly, \(\mathbb{B} \rightarrow \mathbb{B}\) is a tautology, and therefore we have for \(\bar{e}\) being the local variables of \(\tau\) combined with \(\bar{V} \setminus V\):

\[
\begin{align*}
CT & \models \forall (B_1 \land B) \rightarrow \exists y_2 \exists x \exists z. ((\mathbb{L} = \mathbb{L}) \land \mathbb{P} \bowtie \mathbb{P}) \land (B_1 \land B) \land \mathbb{B} \land \mathbb{B}) \land \\
& \forall (B_2 \rightarrow \exists y_2 \exists x \exists z. ((\mathbb{L} = \mathbb{L}) \land \mathbb{P} \bowtie \mathbb{P}) \land (B_2 \land B) \land \mathbb{B} \land \mathbb{B})
\end{align*}
\]

As the local variables of \(\sigma_1 \circ \nu \tau\) are \(\bar{x} \cup \bar{y}_1 \cup \bar{z}\), and analogously for \(\sigma_2 \circ \nu \tau\), we conclude by Thm. 5:

\[
\sigma_1 \circ \nu \tau = \langle L_1 \cup L; P_1 \cup P; B_1 \land B; (V' \cup \bar{V}) \setminus V \rangle \equiv \langle L_2 \cup L; P_2 \cup P; B_2 \land B; (V' \cup \bar{V}) \setminus V \rangle = \sigma_2 \circ \nu \tau.
\]

\(\square\)

An important property for program analysis is monotonicity. The following lemma formulates monotonicity of CHR derivations using the merge operator.

**Lemma 10.4** (Monotonicity). If \([\sigma] \rightarrow_! [\tau] \text{ then } [\sigma] \circ \nu [\sigma'] \rightarrow_! [\tau] \circ \nu [\sigma'] \text{ for all } \forall\).

**Proof.** Let \([\sigma] \rightarrow_! [\tau]\) via rule \(r\) of the form \(H_1 \setminus H_2 \leftrightarrow G \mid B_c, B_0\).

Case **ApplyLinear**: w.l.o.g. let \(\sigma = \langle H_1^! \cup H_1^* \cup L; H_1^P \cup H_1^P ; G \land B \land B; V_1 \rangle\). Then

\[
[\sigma] \rightarrow_! [\langle H_1^! \cup B_c \cup L; H_1^P \cup H_1^P \cup P \cup P'; G \land B \land B; V_1 \rangle]
\]

and therefore for \(\sigma' = \langle L'; P'; B'; V' \rangle\) with w.l.o.g. disjunct local variables:

\[
[\sigma] \circ \nu [\sigma'] = [\sigma \circ \nu \sigma']
\]

\[
\equiv [\langle H_1^! \cup B_c \cup L; H_1^P \cup H_1^P \cup P \cup P'; G \land B \land B; V_1 \rangle]
\]

Case **ApplyPersistent**: w.l.o.g. let \(\sigma = \langle H_1^! \cup L; H_1^P \cup H_1^P \cup P \cup P; G \land B \land B; V_1 \rangle\). Then

\[
[\sigma] \rightarrow_! [\langle H_1^! \cup L; H_1^P \cup H_1^P \cup P \cup B_c; G \land B \land B; V_1 \rangle]
\]

and therefore for \(\sigma' = \langle L'; P'; B'; V' \rangle\) with w.l.o.g. disjunct local variables:

\[
[\sigma] \circ \nu [\sigma'] = [\sigma \circ \nu \sigma']
\]

\[
\equiv [\langle H_1^! \cup L \cup L'; H_1^P \cup H_1^P \cup P \cup P'; G \land B \land B; V_1 \rangle]
\]

By definition, the merge operator is commutative and closed on \(\Sigma_1\). For \(\forall = \emptyset\), the merge operator \(\circ \nu\) is also associative on \(\Sigma_1\), as the following lemma shows. Finally, the state \(\sigma_\emptyset := \langle \emptyset; \emptyset; T; \emptyset \rangle\) is the neutral element of \(\circ\) (for \(\forall = \emptyset\)). In general however, \(\sigma \circ \nu \sigma_\emptyset \neq \sigma\), because \(\sigma\) may have global variables that also occur in \(\forall\).

**Lemma 10.5.** \((\Sigma_1 / \equiv_1, \circ)\) is a commutative monoid.
Proof. Totality of the merge operator and the neutral element \([\sigma_0]\) are clear. Commutativity is inherited from commutativity of the operators used in Def. 10.1, hence, it remains to show associativity of \(\land\):

Let \(\sigma_i = \langle L_i; P_i; B_i; V_i \rangle\) for \(i = 1, 2, 3\), with w.l.o.g. suitably named variables, such that the below \(\land\) operations are defined.

\(\land\) on \(\Sigma\):

\[
(\sigma_1 \land \sigma_2) \land \sigma_3 = (\langle L_1 \cup L_2; P_1 \cup P_2; B_1 \land B_2; V_1 \cup V_2 \rangle \land \sigma_3) = \langle L_1 \cup L_2 \cup L_3; P_1 \cup P_2 \cup P_3; B_1 \land B_2 \land B_3; V_1 \cup V_2 \cup V_3 \rangle = \\
\sigma_1 \land (\sigma_2 \land \sigma_3)
\]

\(\land\) on \(\Sigma/\equiv_1\):

\[
([\sigma_1] \land [\sigma_2]) \land [\sigma_3] = \\
[\sigma_1 \land \sigma_2] \land [\sigma_3] = \\
[(\sigma_1 \land \sigma_2) \land \sigma_3] = \\
[\sigma_1 \land (\sigma_2 \land \sigma_3)] = \\
[\sigma_1] \land ([\sigma_2] \land [\sigma_3])
\]

Example 10.6. For differing \(V\) and \(V'\) we have no associativity for the general case: \((\sigma_1 \land V \sigma_2) \land V' \sigma_3 \neq \sigma_1 \land V (\sigma_2 \land V' \sigma_3)\). This can be seen via the following example:

\[
\begin{align*}
\sigma_1 & = \langle c(X); \emptyset; \top; \{X\} \rangle \\
\sigma_2 & = \langle \emptyset; \emptyset; \top; \emptyset \rangle \\
\sigma_3 & = \langle \emptyset; \emptyset; X = 1; \{X\} \rangle \\
\forall & = \{X\} \\
\forall' & = \emptyset
\end{align*}
\]

\[
\Rightarrow \quad (\sigma_1 \land V \sigma_2) \land V' \sigma_3 = \text{undefined} \\
\sigma_1 \land V (\sigma_2 \land V' \sigma_3) = \langle c(X); \emptyset; X = 1; \emptyset \rangle \\
([\sigma_1] \land [\sigma_2]) \land V' [\sigma_3] = ([c(Y); \emptyset; X = 1; \{X\}] \\
[\sigma_1] \land V ([\sigma_2] \land V' [\sigma_3]) = ([c(X); \emptyset; X = 1; \{X\}])
\]

Nevertheless, we have the following technical lemma:

Lemma 10.7. Let \(\sigma_1, \sigma_2, \sigma_3 \in \Sigma\) such that no local variable of a state occurs in another state. Then it holds for all \(\forall\) that

\[
\sigma_1 \land V (\sigma_2 \land \sigma_3) = (\sigma_1 \land \sigma_2) \land V \sigma_3
\]

and

\[
[\sigma_1] \land V ([\sigma_2] \land [\sigma_3]) = ([\sigma_1] \land [\sigma_2]) \land V [\sigma_3]
\]

Proof. As all local variables of \(\sigma_1, \sigma_2,\) and \(\sigma_3\) are disjunct, the second expression is well-defined and follows directly from the first. Therefore, only the first expression has to be shown. However, this follows directly from the associativity of the operators \(\lor, \land,\) and \(\land\) used in the definition of operator \(\land\) – observe that the global variables of the resulting representant state are \((V_1 \cup V_2 \cup V_3) \setminus V\) in both cases.
10.2 Partial Order on States

As the merge operator (with $V = \emptyset$) is a commutative monoid, there exists an implied preordering. Furthermore, it is also antisymmetric, hence, we arrive at the following partial order.

Lemma 10.8 (Partial Order $\prec$). The commutative monoid $(\Sigma_t/\equiv_t, \circ)$ implies a partial order $\prec$ defined as follows, with $\sigma, \sigma' \in \Sigma$:

$$[\sigma] \prec [\sigma'] \text{ if and only if } \exists [\hat{\sigma}], [\sigma] \circ [\hat{\sigma}] = [\sigma']$$

Proof. Every commutative monoid implies a preorder according to the above definition of $\prec$. Therefore, it suffices to show antisymmetry:

Let $[\sigma_1] \prec [\sigma_2]$ and $[\sigma_2] \prec [\sigma_1]$, then there exist $[\hat{\sigma}_1]$ and $[\hat{\sigma}_2]$ as follows:

$$[\sigma_2] = [\sigma_1] \circ [\hat{\sigma}_1] \quad [\sigma_1] = [\sigma_2] \circ [\hat{\sigma}_2] \quad \Rightarrow [\sigma_1] = ([\sigma_1] \circ [\hat{\sigma}_1]) \circ [\hat{\sigma}_2] = [\sigma_1] \circ ([\hat{\sigma}_1] \circ [\sigma_2])$$

It follows that $[\hat{\sigma}_1] \circ [\hat{\sigma}_2]$ equals $[\sigma_0]$, hence, $[\hat{\sigma}_1] = [\hat{\sigma}_2] = [\sigma_0]$ and therefore, $[\sigma_1] = [\sigma_2]$. $$\square$$

However, special care has to be taken for states that contain local variables, because the resulting partial order may become counter-intuitive, as the following example shows.

Example 10.9. Let $\sigma_1 = \langle c(X); \emptyset; \top; \{X\} \rangle$, $\sigma_2 = \langle c(X); \emptyset; X = 1; \{X\} \rangle$, then $[\sigma_1] \prec [\sigma_2]$ (via $\hat{\sigma} = (\emptyset; \emptyset; X = 1; \{X\})$).

However, for $\sigma_1' = \langle c(X); \emptyset; \top; \emptyset \rangle$ and $\sigma_2' = \langle c(X); \emptyset; X = 1; \emptyset \rangle$ we do not have $[\sigma_1'] \prec [\sigma_2']$. No $\hat{\sigma}$ can exist according to the definition of $\prec$, because the variable restriction in Def. 10.1 for the merge operator $\circ$ prohibits usage of the local variable $X$ in $\hat{\sigma}$.

This partial order combines well with monotonicity: Given that $[\sigma] \prec [\sigma']$ and $[\sigma] \rightarrow_* [\tau]$, we have that $\exists [\hat{\sigma}], [\sigma] \circ [\hat{\sigma}] = [\sigma']$ and $[\sigma'] \rightarrow_* [\tau] \circ [\hat{\sigma}]$. Furthermore, we find that the neutral element $[\sigma_0]$ is also the least element of the partial order.

Excursion: Partial Order $\prec$ in [Duck et al., 2007]

We have found the first occurrence of a partial order on CHR states to be [Duck et al., 2006] and its follow-up version [Duck et al., 2007]. In the context of that work, it is not a partial order on arbitrary CHR states, but relates valid extensions of a common state.

Let us first provide the definitions of extensions and valid extensions from [Duck et al., 2007]. We provide _ for the integer used in $\omega_t$ states, because Duck et al. [2007] work with an operational semantics close to $\omega_t$, but without identified constraints.

Definition 10.10 (Extension). A state $\sigma = \langle G; S; B; T \rangle^V_t$ can be extended by another state $\sigma_e = \langle G_e; S_e; B_e; T_e \rangle^V_e$ as follows

$$\sigma \oplus \sigma_e = \langle G \uplus G_e; S \uplus S_e; B \land B_e; T \uplus T_e \rangle^V_t$$

We say that $\sigma_e$ is an extension of $\sigma$. 

53
Therefore, the extension operation $\oplus$ is defined similarly to our merge operation $\diamond$, differing only in the treatment of global variables. Next, Duck et al. [2007] restrict extensions by introducing the subset of valid extensions according to the following definition.

**Definition 10.11 (Valid Extension).** A valid extension $\sigma_e = (G_e; S_e; B_e; T_e)^{V_e}$ of a state $(G; S; B; T)^V$ is an extension, such that

$$v \in \text{vars}(G, S, B, T) \land v \notin V \Rightarrow v \notin \text{vars}(G_e, S_e, B_e, T_e, V_e).$$

The concept of valid extensions is motivated by Duck et al. [2007] in order to get an extension mechanism that preserves joinability. Using our merge operation we can show that preservation of joinability is a direct consequence of monotonicity in Lemma 10.4 (cf. Cor. 14.9).

Next, in [Duck et al., 2007] a partial order is defined over extensions of a state.

**Definition 10.12 (Partial Order).** Given a state $\sigma = (G; S; B; T)^V$, and valid extensions $\sigma_{e1}$ and $\sigma_{e2}$ of $\sigma$, then we define $\sigma_{e1} \preceq_{\sigma} \sigma_{e2}$ to hold if

1. there exists a valid extension $\sigma_{e3}$ of $(\sigma \oplus \sigma_{e1})$ such that $(\sigma \oplus \sigma_{e1}) \oplus \sigma_{e3} \approx \sigma \oplus \sigma_{e2}$
2. $V - V_{e1} \subseteq V - V_{e2}$ holds.

The authors of this definition state in a footnote that they believe $\preceq_{\sigma}$ to be a partial order, but omitted a formal discussion. We have shown above that a partial order is on states based on merging or extending can indeed be defined. In light of the work of Duck et al. [2007], an attentive reader may wonder why we spent the effort to define and prove another partial order. In this excursion, we answer these doubts by showing that the above relation $\preceq_{\sigma}$ is in fact no partial order at all, but lacks both, transitivity and antisymmetry. This is shown via the following two counter-examples.

**Antisymmetry:** Let $\sigma = (c(X); \emptyset; \top; \emptyset)^{\{X\}}$, $\sigma_1 = (\emptyset; \emptyset; X = 1; \emptyset)^{\{X\}}$, $\sigma_2 = (\emptyset; \emptyset; X \geq 1 \land X \leq 1; \emptyset)^{\{X\}}$. Then $\sigma_1, \sigma_2$ are valid extensions of $\sigma$ and $\sigma_1 \preceq_{\sigma} \sigma_2$ and $\sigma_2 \preceq_{\sigma} \sigma_1$ (by using the extension $(\emptyset; \emptyset; \top; \emptyset)^{\{X\}}$ for the first condition in Definition 10.12). However, $\sigma_1 \neq \sigma_2$ and therefore $\preceq_{\sigma}$ is not antisymmetric.

**Transitivity:** Now let $\sigma = (\emptyset; \emptyset; \top; \emptyset)^{0}$, $\sigma_1 = (c(X); \emptyset; X = 1; \emptyset)^0$, $\sigma_2 = (c(1); \emptyset; \top; \emptyset)^0$, $\sigma_3 = (c(X); \emptyset; X = 1; \emptyset)^{\{X\}}$. Then $\sigma_1, \sigma_2, \sigma_3$ are valid extensions of $\sigma$. As $\sigma$ has no global variables the second condition for Definition 10.12 is always satisfied. It holds that $\sigma_1 \preceq_{\sigma} \sigma_2$ using the extension $(\emptyset; \emptyset; \top; \emptyset)^{\{X\}}$ for the second condition. Furthermore, $\sigma_2 \preceq_{\sigma} \sigma_3$ using the extension $(\emptyset; \emptyset; X = 1; \emptyset)^{\{X\}}$.

By transitivity, $\sigma_1 \preceq_{\sigma} \sigma_3$ should hold, however this requires existence of a valid extension $\sigma_e = (G_e; S_e; B_e; T_e)^{V_e}$ of $(\sigma \oplus \sigma_1) = \sigma_1$ with

$$(c(X); \emptyset; X = 1; \emptyset)^{\emptyset} \oplus \sigma_e \approx \sigma \oplus \sigma_3 = \sigma_3 = (c(X); \emptyset; X = 1; \emptyset)^{\{X\}}$$

To make those states equivalent $X \in V_e$ is required. However, this is a contradiction to $\sigma_e$ being a valid extension of $\sigma_1$. Therefore, $\preceq_{\sigma}$ also lacks transitivity.

## 11 Discussion

### 11.1 Formulations of State Transition System

In this section, we provide an overview of different formulations available for the CHR state transition system. For both, $\omega_e$ and $\omega_l$, we identify three formulations that are so closely
related, that we may freely switch between them. This gives us the freedom to select the most appropriate formulation at any time.

We begin with $\omega_e$, for which Table III.5 shows the different formulation possibilities for $(\Sigma_e, \rightarrow_e)$. For brevity, Table III.5 and the following tables, omit additional textual requirements found in the corresponding definitions, for example, that we consider rule variants with a certain restriction on local variables, or that we assume local variables to be renamed such that the given merge operations are well-defined.

**Variant I** was our first formulation of the equivalence-based operational semantics, given in Definition 8.6. Its **Equivalence** rule resembles the compliance of state equivalence with rule application given by Theorem 4. Reducing the **Apply** rule, such that it is based on a rule state, yields **Variant I_o**. While it makes **Apply** more compact, this comes at the cost of a third inference rule to handle merges. This **Merge** rule makes monotonicity explicitly part of the transition system (cf. Lemma 10.4). The implicitly required state splitting operation is discussed in more detail in Section 13.3.

Abstracting over state equivalence yields **Variant II** in Table III.6, which is based on equivalence classes. Due to directly rewriting the set of all equivalent states, it no longer requires the **Equivalence** rules from Table III.5. Finally, **Variant II_o** analogously makes monotonicity explicit via the **Merge** rule.

Clearly, all formulations given in Table III.5 and Table III.6 are sound and complete with respect to each other and $\omega_{va}$. For the remainder of this work, we will mostly apply **Variant II**, as it is the most succinct formulation. When discussing program analysis methods, however, we often refer to **Variant II_o** due to the importance of the merge operator in that context. We further found the direct rewriting of equivalence classes, instead of their single representant states, to be beneficial in the context of this work. Hence, the first two variants are rarely applied in the remaining chapters.

Analogously to Table III.5 and Table III.6, we list different formulations of the operational semantics $\omega_!$ in Table III.7 and Table III.8. We again find Theorem 4 in the form of the **Equivalence** rule and Lemma 10.4 corresponds to the **Merge** rule.

### 11.2 Range-Restrictedness

As specified in Definition 9.5, $\omega_!$ requires range-restricted programs. In the following, we explain why a naive extension to the full segment of CHR by dropping this restriction would violate soundness.

We recall that a persistent constraint is a finite representation of an arbitrary number of identical constraints, as generated under $\omega_e$ by a propagation rule from the range-restricted segment. Under the same conditions, however, a propagation rule with local variables would generate an arbitrary number of nearly but not quite identical constraints, as the local variables would be renamed apart between any two of those nearly identical constraints, which the following example demonstrates.

**Example 11.1.** Consider the following program:

\[
\begin{align*}
    r_1 & \quad @ \quad a \quad \Rightarrow \quad b(X) \\
    r_2 & \quad @ \quad b(X), b(X) \quad \Leftrightarrow \quad c
\end{align*}
\]

When executed with an initial $a$-constraint, this program causes the following infinite derivations.
11. Discussion

Variant I: \((\Sigma_e, \rightarrow_e)\)

Apply:

\[
\begin{align*}
& r @ H_1 \setminus H_2 \Leftrightarrow G | B_c, B_b \\
& (H_1 \uplus H_2 \uplus G; G \land B; V) \rightarrow^e_e (H_1 \uplus B_c \uplus G; G \land B_b \land B; V)
\end{align*}
\]

Equivalence:

\[
\begin{align*}
\sigma' \equiv \sigma & \quad \sigma \xrightarrow{r} \tau \\
\tau' \equiv \tau & \quad \sigma' \xrightarrow{r_e} \tau'
\end{align*}
\]

Variant I\(\circ\): \((\Sigma_e, \rightarrow_e)\) with \(\circ\)

Apply:

\[
\begin{align*}
& r @ H_1 \setminus H_2 \Leftrightarrow G | B_c, B_b \\
& (H_1 \uplus H_2; G; V) \rightarrow^e_e (H_1 \uplus B_c; G \land B_b; V)
\end{align*}
\]

Equivalence:

\[
\begin{align*}
\sigma' \equiv \sigma & \quad \sigma \xrightarrow{r} \tau \\
\tau' \equiv \tau & \quad \sigma' \xrightarrow{r_e} \tau'
\end{align*}
\]

Merge:

\[
\begin{align*}
\sigma \xrightarrow{r_e} \tau & \quad \sigma \circ \delta \xrightarrow{r_e} \tau \circ \delta
\end{align*}
\]

Table III.5: Different Formulations of the Operational Semantics \(\omega_e\) over \(\Sigma_e\)

Variant II: \((\Sigma_e/\equiv_e, \rightarrow_e)\)

Apply:

\[
\begin{align*}
& r @ H_1 \setminus H_2 \Leftrightarrow G | B_c, B_b \\
& [(H_1 \uplus H_2 \uplus G; G \land B; V)] \rightarrow^e_e [(H_1 \uplus B_c \uplus G; G \land B_b \land B; V)]
\end{align*}
\]

Variant II\(\circ\): \((\Sigma_e/\equiv_e, \rightarrow_e)\) with \(\circ\)

Apply:

\[
\begin{align*}
& r @ H_1 \setminus H_2 \Leftrightarrow G | B_c, B_b \\
& [(H_1 \uplus H_2; G; V)] \rightarrow^e_e [(H_1 \uplus B_c; G \land B_b; V)]
\end{align*}
\]

Merge:

\[
\begin{align*}
|\sigma| \xrightarrow{e} |\tau| & \quad |\sigma| \circ \delta \xrightarrow{e} |\tau| \circ \delta
\end{align*}
\]

Table III.6: Different Formulations of the Operational Semantics \(\omega_e\) over \(\Sigma_e/\equiv_e\)
Variant I: \((\Sigma_1, \rightarrow_1)\)

ApplyLinear:
\[
\begin{array}{l}
\frac{r \otimes (H_1^l \cup H_1^p) \setminus (H_2^l \cup H_2^p) \Leftrightarrow G \mid B_c, B_b \quad H_2^l \neq \emptyset \quad \sigma \neq \tau}{\sigma = \langle H_1^l \cup H_2^l \cup L; H_1^p \cup H_2^p \cup P; G \land B; V \rangle} \\
\rightarrow^r \langle H_1^l \cup B_c \cup L; H_1^p \cup H_2^p \cup P; G \land B \land B_b; V \rangle = \tau
\end{array}
\]

ApplyPersistent:
\[
\begin{array}{l}
\frac{r \otimes (H_1^l \cup H_1^p) \setminus H_2^p \Leftrightarrow G \mid B_c, B_b \quad \sigma \neq \tau}{\sigma = \langle H_1^l \cup L; H_1^p \cup H_2^p \cup P; G \land B; V \rangle} \\
\rightarrow^r \langle H_1^l \cup L; H_1^p \cup H_2^p \cup B_c \cup P; G \land B \land B_b; V \rangle = \tau
\end{array}
\]

Equivalence:
\[
\begin{array}{l}
\frac{\sigma' \equiv \sigma \quad \sigma \rightarrow^r \tau \quad \tau \equiv \tau'}{\sigma' \rightarrow^e \tau'}
\end{array}
\]

Variant \(I_0\): \((\Sigma_1, \rightarrow_1)\) with \(\diamond\)

ApplyLinear:
\[
\begin{array}{l}
\frac{r \otimes (H_1^l \cup H_1^p) \setminus (H_2^l \cup H_2^p) \Leftrightarrow G \mid B_c, B_b \quad H_2^l \neq \emptyset \quad \sigma \neq \tau}{\sigma = \langle H_1^l \cup H_2^l; H_1^p \cup H_2^p; G; V \rangle} \\
\rightarrow^r \langle H_1^l \cup B_c; H_1^p \cup H_2^p; G \land B_b; V \rangle = \tau
\end{array}
\]

ApplyPersistent:
\[
\begin{array}{l}
\frac{r \otimes (H_1^l \cup H_1^p) \setminus H_2^p \Leftrightarrow G \mid B_c, B_b \quad \sigma \neq \tau}{\sigma = \langle H_1^l; H_1^p \cup H_2^p; G; V \rangle} \\
\rightarrow^r \langle H_1^l; H_1^p \cup H_2^p \cup B_c; G \land B_b; V \rangle = \tau
\end{array}
\]

Equivalence:
\[
\begin{array}{l}
\frac{\sigma' \equiv \sigma \quad \sigma \rightarrow^r \tau \quad \tau \equiv \tau'}{\sigma' \rightarrow^e \tau'}
\end{array}
\]

Merge:
\[
\frac{\sigma \rightarrow^r \tau}{\sigma \diamond \delta \rightarrow^r \tau \diamond \delta}
\]

Table III.7: Different Formulations of the Operational Semantics \(\omega_1\) over \(\Sigma_1\)
11. Discussion

**Variant II:** $(\Sigma_! \equiv !, \rightarrow_I)$

**ApplyLinear:**
\[
  r \circ (H_1^I \cup H_1^P) \setminus (H_2^I \cup H_2^P) \Rightarrow G \mid B_c, B_b \quad H_2^I \neq \emptyset \quad [\sigma] \neq [\tau]
\]
\[
  [\sigma] = [(H_1^I \cup H_2^I \cup P; G \land B \land V)]
\]
\[
  \Rightarrow [\sigma] = [(H_1^I \mid B_c \cup P; G \land B \land V)] = [\tau]
\]

**ApplyPersistent:**
\[
  r \circ (H_1^I \cup H_1^P) \setminus H_2^P \Rightarrow G \mid B_c, B_b \quad [\sigma] \neq [\tau]
\]
\[
  [\sigma] = [(H_1^I \cup P; H_2^P \cup H_2^I \cup P; G \land B \land V)]
\]
\[
  \Rightarrow [\sigma] = [(H_1^I \mid B_c \cup H_2^I \cup P; G \land B \land V)] = [\tau]
\]

**Variant II.5:** $(\Sigma_! \equiv !, \rightarrow_I)$ with $\diamond$

**ApplyLinear:**
\[
  r \circ (H_1^I \cup H_1^P) \setminus (H_2^I \cup H_2^P) \Rightarrow G \mid B_c, B_b \quad H_2^I \neq \emptyset \quad [\sigma] \neq [\tau]
\]
\[
  [\sigma] = [(H_1^I \cup H_2^I \cup P; G \land V)]
\]
\[
  \Rightarrow [\sigma] = [(H_1^I \mid B_b \cup H_1^I \cup P; G \land V)] = [\tau]
\]

**ApplyPersistent:**
\[
  r \circ (H_1^I \cup H_1^P) \setminus H_2^P \Rightarrow G \mid B_c, B_b \quad [\sigma] \neq [\tau]
\]
\[
  [\sigma] = [(H_1^I \mid B_c \cup H_1^I \cup H_2^I \cup P; G \land V)]
\]
\[
  \Rightarrow [\sigma] = [(H_1^I \mid H_2^I \cup B_c \cup H_2^P; G \land V)] = [\tau]
\]

**Merge:**
\[
  [\sigma] \Rightarrow [\tau]
\]
\[
  [\sigma] \triangleright \llbracket \delta \rrbracket \Rightarrow [\tau] \triangleright \llbracket \delta \rrbracket
\]

Table III.8: Different Formulations of the Operational Semantics $\omega_I$ over $\Sigma_! \equiv !$
tion under \( \omega_e \).

\[
\begin{align*}
&\vdash_{\omega_e} \left[ (a; \top; \emptyset) \right] \\
&\vdash_{\omega_e}^1 \left[ (a, b(X'); \top; \emptyset) \right] \\
&\vdash_{\omega_e}^2 \left[ (a, b(X'), b(\langle X' \rangle); \top; \emptyset) \right] \vdash_{\omega_e}^* \ldots
\end{align*}
\]

The variables \( X', X'', \ldots \) are distinct from each other and from the variable \( X \) which occurs in the rule body. Thus, it is impossible to derive the \( c \)-constraint from the \( a \)-constraint under \( \omega_e \).

Under the current approach, we cannot finitely represent an arbitrary number of such nearly identical constraints. A naive extension of \( \omega_e \) to the full segment of CHR as specified above would discard the distinction between the two types of generated constraints altogether.

**Example 11.2.** With respect to the previous example, a naive extension of \( \omega_e \) would make the following derivation possible:

\[
\begin{align*}
&\vdash_{\omega_e} \left[ (a; \emptyset; \top; \emptyset) \right] \\
&\vdash_{\omega_e}^1 \left[ (a; b(X'); \top; \emptyset) \right] = \left[ (a; b(X'), b(X'); \top; \emptyset) \right] \\
&\vdash_{\omega_e}^2 \left[ (a; b(X'), b(X'), c; \top; \emptyset) \right]
\end{align*}
\]

As the above example showed, simply applying Definition 9.5 to non-range-restricted programs results in a loss of soundness.

### 11.3 Termination Behavior

The operational semantics \( \omega_i \) is an approach to treating propagation rules in CHR that is completely different from the previous token-based approaches. This especially results in a differing termination behavior. There exist programs that terminate in \( \omega_i \) but not in \( \omega_e \) and vice versa.

**Example 11.3.** Consider again the example program for computing the transitive hull, given in Example 9.1. Due to the presence of a propagation rule, it is non-terminating under \( \omega_e \). Under \( \omega_i \) and \( \omega_p \), termination depends on the initial goal: It is shown in Pilozzi and De Schreye [2009] that it terminates for acyclic graphs. However, goals containing cyclic graphs, such as \( \langle e(1, 2), e(2, 1); \emptyset; \top; \emptyset \rangle_0 \), entail non-terminating behavior. The following derivation is not exact according to \( \omega_i \), but uses simplified states for better readability.

\[
\begin{align*}
&\vdash_t \langle e(1, 2), e(2, 1); \emptyset; \top; \emptyset \rangle_0^0 \\
&\vdash_t \langle \emptyset; e(1, 2)\#0, e(2, 1)\#1; \top; \emptyset \rangle_2^0 \\
&\vdash_t \langle e(1, 1); e(1, 2)\#0, e(2, 1)\#1; \top; \{ (t, 0, 1) \} \rangle_2^0 \\
&\vdash_t \langle \emptyset; e(1, 2)\#0, e(2, 1)\#1, e(1, 1)\#2; \top; \{ (t, 0, 1) \} \rangle_3^0 \\
&\vdash_t \langle e(1, 2); e(1, 2)\#0, e(2, 1)\#1, e(1, 1)\#2; \top; \{ (t, 0, 1), (t, 2, 0) \} \rangle_3^0 \\
&\vdash_t \ldots
\end{align*}
\]

Under \( \omega_i \), the previous goal terminates after computing the transitive hull.

\[
\begin{align*}
&\vdash_{\omega_i} \left[ \langle e(1, 2), e(2, 1); \emptyset; \top; \emptyset \rangle \right] \\
&\vdash_{\omega_i}^* \left[ \langle e(1, 2), e(2, 1); \{ e(1, 1) \}; \top; \emptyset \rangle \right] \\
&\vdash_{\omega_i}^* \left[ \langle e(1, 2), e(2, 1); \{ e(1, 1), e(1, 2), e(2, 1), e(2, 2) \}; \top; \emptyset \rangle \right] \not\vdash_{\omega_i}
\end{align*}
\]
The transitive hull program benefits from execution under $\omega_t$. While it is the most natural formulation of a transitivity property in terms of a CHR rule, current implementations cannot use it in that form due to its non-terminating behavior on circular input graphs. For $\omega_t$, we can instead prove termination for arbitrary inputs.

**Lemma 11.4.** Under $\omega_t$, the transitive hull program terminates for every possible input.

**Proof.** The only rule propagates constraints of type $e/2$, which necessarily become persistent. The propagated constraints contain only the arguments $X,Z$, received as arguments in the rule head. Hence, no new arguments are introduced. Any given initial state contains a finite number of arguments used in $e/2$ constraints. From these, only finitely many different $e/2$ constraints can be built. As rule application is irreflexive, the computation therefore has to stop after a finite number of transition steps.

Nevertheless, program termination in $\omega_t$ is not strictly stronger than that in $\omega_t$ or $\omega_p$, as the following counterexample shows.

**Example 11.5.** Consider the following exemplary CHR program.

\[
\begin{align*}
  r_1 & @ a \quad \Longrightarrow \quad b \\
  r_2 & @ c(X), b \quad \leftarrow \quad c(X+1)
\end{align*}
\]

The program terminates in $\omega_t$ (and $\omega_p$): As there can only be a finite number of $a$-constraints in the initial goal, rule $r_1$ will create a finite number of $b$-constraints as well. These will be consumed by rule $r_2$ in finite time, followed by quiescence. We again simplified the following states for better readability.

\[
\begin{align*}
  & (a,c(X); \emptyset; \top; \emptyset\{X\}) \\
  \Rightarrow_t & (\emptyset; a\#0, b\#1, c(X)\#2; \top; (r_1, 0)\{X\}) \\
  \Rightarrow_t & (c(X+1); a\#0; \top; (r_1, 0)\{X\}) \\
  \Rightarrow_t & (\emptyset; a\#0, c(X+1)\#3; \top; (r_1, 0)\{X\}) \\
  \not\Rightarrow_t & (a,c(X); \emptyset; \top; \emptyset\{X\})
\end{align*}
\]

In contrast, the same program exhibits non-terminating behavior in $\omega_t$, as the following infinite derivation shows:

\[
\begin{align*}
  & [(a,c(X); \emptyset; \top; \emptyset\{X\})] \\
  \not\Rightarrow_t & [(a,c(X); b; \top; \emptyset\{X\})] \\
  \not\Rightarrow_t & [(a,c(X+1); b; \top; \emptyset\{X\})] \\
  \not\Rightarrow_t & [(a,c(X+2); b; \top; \emptyset\{X\})] \\
  \not\Rightarrow_t & \ldots
\end{align*}
\]

This difference in termination behavior is neither good nor bad. It only shows that our approach of persistent constraints is an entirely different one from the token-based approaches. Therefore, programs developed for either operational semantics should not naively be executed in the other, lest results may change unexpectedly.

### 11.4 Expressivity

*Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.*

— Alan Jay Perlis (1922–1990), Computer Scientist
In this section we compare expressivity of the operational semantics \(\omega_e, \omega_l, \omega_p\), and \(\omega_l\). Section 11.4.1 first introduces how we formally compare expressivity of different operational semantics, before Section 11.4.2 presents the results of our comparison.

11.4.1 Expressivity of Operational Semantics

As all of the compared operational semantics are Turing-complete [Sneyers et al., 2009b], expressivity is compared in the literature via the concept of acceptable encoding. This concept originates from Shapiro [Shapiro, 1989] and was first applied to CHR by Gabbrielli et al. [2009]. It relies on the notion of answer defined below.

In order to distinguish linear and persistent constraints when considering goals, we introduce for each CHR constraint symbol \(c/n\), denoting a linear constraint, a corresponding fresh symbol \(!c/n\), denoting a persistent constraint. For a multiset \(M = \{c_1(t_1), \ldots, c_n(t_n)\}\) let \(!M = \{!c_1(t_1), \ldots, !c_n(t_n)\}\).

In the literature answers are usually defined as logical formulas, expressing the declarative reading of a final state. We found it more suitable to define them as \(\omega_e\) states for two reasons: Firstly, unlike logical formulas, \(\omega_e\) states are aware of multiplicities of constraints. Secondly, \(\omega_e\) states enable us to exploit \(\equiv_e\) when comparing answers.

**Definition 11.6 (Answers).** Let \(G \land B\) be a goal with CHR constraints \(G\) and built-in constraints \(B\). Then the set of equivalence classes of \(\omega_e\) states \(A_P(G \land B)\) for a program \(P\) is called the (set of) answers and is defined as follows:

- for \(\omega_e\): \(A_P^e(G \land B) = \{[\tau] \mid [(G; B; \text{vars}(G \land B))] \equiv^e \tau \not\equiv^e \}\)
- for \(\omega_l\): \(A_P^l(G \land B) = \{[(\text{chr}(G); B; \text{vars}(G \land B))] \mid (G, B; \emptyset; \top; \emptyset)_{\text{vars}(G \land B)} \equiv^l \langle \emptyset; G; B; T \rangle_{\text{vars}(G \land B)} \not\equiv^l \}\)
- for \(\omega_p\): \(A_P^p\) is defined analogously to \(A^l\).
- for \(\omega_l\): \(A_P^l(G \land B) = \{[(L^0[P]; B; \text{vars}(G \land B))] \mid G = L^0[P \land (L; P; B; \text{vars}(G \land B)) \equiv^l (L; P; B; \text{vars}(G \land B)) \not\equiv^l \}\)

The following definition is based on the definition of Gabbrielli et al. [2009] for an acceptable encoding for CHR operational semantics.

**Definition 11.7 (Acceptable Encoding).** Let \(\omega_1, \omega_2\) be two operational semantics, \(P_i\) the set of all \(\omega_i\) programs, and \(G_i\) the set of all \(\omega_i\) goals for \(i = 1, 2\). An acceptable encoding of \(\omega_1\) into \(\omega_2\) is a pair of mappings \([\ ] : P_1 \rightarrow P_2\) and \([\ ]_g : G_1 \rightarrow G_2\) which satisfy the following conditions:

- \(P_1\) and \(P_2\) share the same constraint theory \(CT\);
- for any goal \(G \in G_1\) let \(c, d \in G\) be CHR constraints, then \([c, d]_g = [c]_g \uplus [d]_g\). For any built-in constraint \(b \in G\) we have \([b]_g = b\).
- Answers are preserved, i.e., \(\forall G \in G_1. \forall P \in P_1. A^2_P([G]_g) = [A^1_P(G)]_g\) holds.
11. Discussion

11.4.2 Comparison Results

Figure III.1 orders the different operational semantics by expressivity. As shown by Gabbrielli et al. [2009], there exists an acceptable encoding to embed $\omega_t$ into $\omega_p$, but not vice versa. Thus, $\omega_p$ is strictly more expressive than $\omega_t$, denoted by the corresponding arrow in Figure III.1. In this work, we furthermore show that $\omega_p$ is strictly more expressive than $\omega!$ and that $\omega_e$ is strictly less expressive than both $\omega_t$ and $\omega!$.

Concerning the embedding of $\omega_e$ into $\omega!$, we assume range-restricted programs only. Concerning the acceptable encodings of $\omega!$ into $\omega_t$ and $\omega_p$, we require that the respective programs do not contain pathological rules, according to the following definition.

Definition 11.8 (Pathological Rules). A CHR rule

$$r @ H_1 \iff H_2 \Leftrightarrow G \mid B_c, B_b$$

is called pathological if and only if

$$\exists B . \langle H_2; B \land G; \emptyset \rangle \equiv_e \langle B_c; B_b; \emptyset \rangle$$

It is called trivially pathological iff $B = \top$. A CHR program $\mathcal{P}$ is called pathological if it contains at least one pathological rule.

The range-restriction requirement on $\omega_e$ programs is due to the fact that Definition 9.5 for $\omega!$ is only defined on range-restricted programs. The restriction to non-pathological programs for embeddings of $\omega_t$ into $\omega!$ and $\omega_p$ ensures **ApplyLinear** transitions never fail due to irreflexivity, according to the following Lemma.

Concerning the relationship of $\omega_t$ and $\omega!$, we found that no acceptable encoding of $\omega_t$ into $\omega!$ exists. We did find an acceptable encoding of $\omega!$ into $\omega_t$. However, a thusly encoded program might exhibit a different termination behavior from the original $\omega!$ program (cf. Example 11.16), as visualized by the dashed arrow in Figure III.1. We currently do not know whether an acceptable encoding without that limitation exists.

The definition of pathological rules is chosen such as to coincide with those rules that cause redundant rule applications – modulo state equivalence – in $\omega_e$.

Lemma 11.9. Let $\mathcal{P}$ be a non-pathological CHR program. Then for all $\omega_e$ states $\sigma, \tau \in \Sigma_e$ where $[\sigma] \rightarrow_e [\tau]$, we have $\sigma \neq_e \tau$.

Proof. We first show a property of Def. 11.8: Let $\langle H_2; B \land G; \emptyset \rangle \equiv_e \langle B_c; B_b; \emptyset \rangle$, w.l.o.g. let the respective local variables $\bar{y}, \bar{y}'$ be renamed apart. Then by Thm. 3:

$$CT \models \forall (B \land G \rightarrow \exists \bar{y}'.((H_2 = B_c) \land B_b))$$

and

$$CT \models \forall (B_b \rightarrow \exists \bar{y}.((H_2 = B_c) \land B \land G))$$
This is logically equivalent to

\[
CT \models \forall (B \land G \to \exists y'.((H_2 = B_c) \land B_b \land B \land G)) \text{ and } \\
CT \models \forall (B_b \land B \land G \to \exists y,.((H_2 = B_c) \land B \land G))
\]

Therefore, again by Thm. 3, we have that

\[
\langle H_2; G \land B; \emptyset \rangle \equiv_e \langle B_c; B_b; \emptyset \rangle \equiv_e \langle B_c; B_b \land G \land B; \emptyset \rangle
\]

Now let \( r \) be a rule \( r \in H_1 \setminus H_2 \iff G \mid B_c, B_b \) such that \( [\sigma] \rightarrow^r \tau \). It follows that \( \sigma \equiv_e \langle H_1 \uplus H_2 \uplus \mathbb{G}; G \land B \land \mathbb{V} \rangle \) and \( \tau \equiv_e \langle B_c \uplus H_1 \uplus \mathbb{G}; B_b \land G \land B \land \mathbb{V} \rangle \).

Assume that \( \sigma \equiv_e \tau \). As \( H_1 \) and \( \mathbb{G} \) occur in both states, the corresponding states with those multisets removed are also equivalent. Similarly, the same states with \( \emptyset \) instead of \( \mathbb{V} \) for global variables are equivalent. Therefore,

\[
\langle H_2; G \land B; \emptyset \rangle \equiv_e \langle B_c; B_b \land G \land B; \emptyset \rangle
\]

This implies that there exists a \( \mathbb{B} \) according to Def. 11.8, which is a contradiction to the program being non-pathological. Hence, \( \sigma \neq_e \tau \). □

The following lemmata are proofs for the arrows present in Figure III.1.

**Lemma 11.10** (\( \omega_l \rightarrow \omega_p \)). There exists an acceptable encoding of \( \omega_l \) into \( \omega_p \).

*Proof (sketch).* Set all rules to priority 1. □

**Lemma 11.11** (\( \omega_p \not\rightarrow \omega_l \)). There exists no acceptable encoding of \( \omega_p \) into \( \omega_l \).

*Proof.* Follows directly from Gabbrielli et al. [2009]. □

**Lemma 11.12** (\( \omega_e \rightarrow \omega_l \)). There exists an acceptable encoding of \( \omega_e \) into \( \omega_l \).

*Proof (sketch).* Replace propagation rules with simplification rules that contain a copy of the head in their bodies. □

**Lemma 11.13** (\( \omega_l \not\rightarrow \omega_e \)). There exists no acceptable encoding of \( \omega_l \) into \( \omega_e \).

*Proof.* For any program \( P' \) if \( \sigma = \langle G'; B'; \emptyset \rangle \) with [\( \sigma \)] \( \in \mathcal{A}_{P'}(G) \), no rule in \( P' \) is applicable to [\( \sigma \)]. Consider the \( \omega_l \) program \( P = (a \Rightarrow b) \). Since \( \mathcal{A}_{P}(a) = \{[[a, b; \top; \emptyset]]\} \) and \( \mathcal{A}_{P}(a, b) = \{[[a, b; \top; \emptyset]]\} \), an acceptable encoding has to satisfy \( \mathcal{A}_{P}(g) = \{[[a, b; \top; \emptyset]]\} \) and \( \mathcal{A}_{P}(g) = \{[[a, b; \top; \emptyset]]\} \neq \{[[a, b; \top; \emptyset]]\} \) which contradicts our earlier observation. □

**Lemma 11.14** (\( \omega_l \not\rightarrow \omega_l \)). There exists no acceptable encoding of \( \omega_l \) into \( \omega_l \).

*Proof.* Analogously to Lemma 11.13. □

**Lemma 11.15** (\( \omega_l \rightarrow \omega_l \)). There exists an acceptable encoding of \( \omega_l \) into \( \omega_l \).
11. Discussion

Proof. We show how to encode any $\omega_1$ program $P$ in $\omega_t$. For every $n$-ary constraint $c/n$ in $P$, there exists an $(n + 1)$-ary constraint $c/n + 1$ in the encoding. In the following, for a multiset of user-defined $\omega_t$-constraints $M = \{c_1(t_1), \ldots , c_n(t_n)\}$ let $l(M) := \{c_1(l(t_1), \ldots , c_n(l t_n))$ and $p(M) := \{c_1(p(t_1), \ldots , c_n(p t_n))\}$.

The encoded program $[P]$ is constructed as follows:

1. For every rule $r \ni H_1 \setminus H_2 \Rightarrow G \mid B$ in $P$, and all multisets $H_1', H_1'', H_2', H_2''$ s.t. $H_1' \uplus H_1'' = H_1$ and $H_2' \uplus H_2'' = H_2$ and $H_2' \neq \emptyset$, the following rule is in $[P]$:

$$l(H_1') \uplus p(H_1'') \uplus p(H_2')) \setminus l(H_2') \Rightarrow G \mid l(B_c), B_b$$

2. For every rule $r \ni H_1 \setminus H_2 \Rightarrow G \mid B_c, B_b$ in $P$, and all multisets $H_1', H_1''$ s.t. $H_1' \uplus H_1'' = H_1$, the following rule is in $[P]$:

$$l(H_1') \uplus p(H_1'') \Rightarrow G \mid p(H_2), B_b$$

3. For every rule $\{c(p, t), c(p, t')\} \uplus H_1 \setminus H_2 \Rightarrow G \mid B$ in $[P]$, add also the following rule:

$$\{c(p, t)\} \uplus H_1 \setminus H_2 \Rightarrow t = t' \land G \mid B$$

4. For every user-defined constraint declaration $c_n$ in $P$, there is a rule

$$c(p, t) \setminus c(p, t) \Rightarrow \top$$

The translation of goals is defined as:

$$[L \uplus !P]_g := l(L) \uplus p(P)$$

Soundness: Let $S_l$ be a function mapping from $\omega_t$ states to $\Sigma$ such that for $\sigma_t = \langle l(L) \uplus p(P) \uplus B' ; S ; B ; T \rangle_k^Y$ where $\text{chr}(S) = l(L') \uplus p(P')$ for some $L', P'$,

$$S_l(\sigma_t) := [L \uplus L' ; P \uplus P' ; B \land B' ; V]$$

In the following, we will show that for all $\sigma_t, \tau_t \in \Sigma$, $\sigma_t \not\rightarrow^* \tau_t$ implies $S_l(\sigma_t) \not\rightarrow^* \tau_l(\tau_t)$.

It is clear from the definition that both the Introduce and Solve transitions of $\omega_t$ are invariant to the $S_l$ function. Concerning Apply, we proceed stepwise w.r.t. the application of the four types of rules present in the encoding $[P]$.

1. The rules introduced in construction step 1 represent $\text{ApplyLinear}$ transitions in $P$.

Let $r$ be a variant of a rule $l(H_1') \uplus p(H_1'') \uplus p(H_2') \setminus l(H_2') \Rightarrow G \mid l(B_c) \uplus B_b$ in $[P]$ with fresh variables $\bar{y}$. By definition of the encoding, $r$ has a corresponding rule $r' \ni H_1' \uplus H_1'' \setminus H_2' \uplus H_2'' \Rightarrow G \mid B_c, B_b$ in $P$. We assume w.l.o.g. that the goal store of $\sigma_t$ is empty. Hence let $\sigma_t = \langle \emptyset ; l(L) \uplus p(P) ; B ; T \rangle_k^Y$ and assume that $\sigma_t \not\rightarrow^* \tau_l$. From Def. 5.9 follows that $CT \models \forall \bar{y}.(l(H_1') \uplus p(H_1'') \uplus p(H_2') \uplus p(H_2'') \uplus p(P') = \emptyset)$ for some $L', P'$.

Hence, $S_l(\sigma_t) = \langle [H_1' \uplus H_1'' \uplus L' ; H_1'' \uplus H_2' \uplus P'] ; (H_1' \uplus H_2'' \uplus P' = \emptyset) \land (H_1' \uplus H_2'' \uplus P' = \emptyset) \land G \mid B ; V \rangle$.

Using Def. 5.9 and Def. 9.5, we now have that $\sigma_l(\sigma_l) \not\rightarrow^* \tau_l(\sigma_l)$ or $S_l(\sigma_l) = \tau_l(\sigma_l)$.

2. The rules introduced in step 2 represent $\text{ApplyPersistent}$ transitions. Analogously to step 1, we have that $\sigma_t \not\rightarrow^* \tau_l$ implies $\tau_l(\sigma_l) \not\rightarrow^* \tau_l(\sigma_l)$ for some rule $r' \in P$ or $S_l(\sigma_l) = \tau_l(\sigma_l)$.

3. Step 3 introduces further rules for both $\text{ApplyLinear}$ and $\text{ApplyPersistent}$ transitions where a single persistent constraint in the store matches with several head constraints.
For example, the rule \(c(X), c(Y) \iff d(X, Y)\) is applicable to the state \([\emptyset; c(0); \top; \emptyset]\) in \(\omega_1\), since \([\emptyset; c(0); \top; \emptyset]\) \(\equiv [\emptyset; c(0), c(0); \top; \emptyset]\). Step 2 of the embedding introduces the rule \(c(p, X), c(p, Y) \iff d(p, X, Y)\) and step 3 furthermore introduces \(c(p, X) \iff X = Y [d(p, X, Y)]\), which matches with the \(\omega_1\) state \([\emptyset; c(p, 0); \top; \emptyset]\). Strengthening of the guard might result in a redundant rule: For the rule \(c(X), c(Y) \iff X > Y [d(p, X, Y)]\), the rule \(c(p, X) \iff \bot [d(p, X, Y)]\) is introduced which cannot be fired by definition.

To proof soundness, let \(\sigma = [\langle L; (c(p, \bar{t}), c(p, \bar{t}^2)) \cup \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle]_2\) and \(\sigma' = [\langle L; (c(p, \bar{t}^2)) \cup \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle]_2\) such that \([\sigma] \rightarrow^* [\tau]\) for some \(\tau\). If \(CT \models \forall (B \rightarrow \bar{t} = \bar{t}^2)\), we have \(\sigma \equiv \bar{\sigma}'\), and hence \([\bar{\sigma}'] \rightarrow^* [\tau]\). The soundness of the rules introduced in step 3 is thus reduced to the soundness of those from step 1 and step 2.

4. The rules introduced in step 4 enforce a minimal representation of the persistent store. As \(S_1((G; \{c(t), (c(t))\} \cup \mathbb{P}; \mathbb{B}; \mathbb{V})) = S_1((G; \{c(t)) \cup \mathbb{P}; \mathbb{B}; \mathbb{V}))\), they are invariant to soundness.

Now assume that \(\sigma = [\langle L; (c(t), c(t)) \cup \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle]_2\) for any goal \(G, B\) and \(\mathbb{B}, \mathbb{V}\). We assume w.l.o.g. that \(\mathbb{L} \neq \emptyset\) and \(\bar{y}, \bar{y}'\) are local variables of \(\sigma, \bar{\sigma}'\). We assume w.l.o.g. that \(\bar{y}, \bar{y}'\) are disjoint. Hence, Thm. 5 implies:

\[
CT \models \forall (B \rightarrow \exists \bar{y}' \cdot ((L = H_1^i \cup H_2^i \cup L') \land (P \bowtie H_4^i \cup H_2^p \cup \mathbb{P}' \land \mathbb{B}' \land G)))
\]

\[
CT \models \forall (B' \land G) \rightarrow \exists \bar{y}' \cdot ((L = H_1^i \cup H_2^i \cup L') \land (P \bowtie H_4^i \cup H_2^p \cup \mathbb{P}' \land \mathbb{B}' \land G)))
\]

By step 1 of our encoding, \([\mathbb{P}]\) contains a rule

\[r' \bowtie l(H_1^i) \cup p(H_1^p) \cup p(H_2^p) \setminus l(H_2^i) \iff G \mid l(B_c), B_b\]

We aptly decompose \(L\) into three components \(L = H_1'^i \cup H_2'^i \cup \mathbb{L}'\) such that:

\[L = H_1'^i \cup H_2'^i \cup \mathbb{L}' \Rightarrow (H_1'^i = H_1^i) \land (H_2'^i = H_2^i) \land (\mathbb{L}' = \mathbb{L}')\]
Step 3 then guarantees that \( \mathcal{P} \) contains a rule

\[
\tau'' \in l(H_1^t) \cup p(H^p) \setminus l(H_2^t) \Leftrightarrow G' \land G \mid l(B_c), B_b
\]

such that

\[
\mathcal{C} \models G' \leftrightarrow H^p \triangleright H_1^t \cup H_2^t
\]

and

\[
\mathcal{P} \triangleright H_1^t \cup H_2^t \cup P' \Rightarrow H^{p'} = H^p
\]

Applying (III.12), (III.10), and (III.11) gives us

\[
\mathcal{C} \models (\mathcal{P} \triangleright H_1^t \cup H_2^t \cup P') \rightarrow G'
\]

Hence, from (III.7), we get

\[
\mathcal{C} \models \forall (\exists \bar{y}'.(\exists (L = H_1^t \cup H_2^t \cup L') \land (\mathcal{P} \triangleright H_1^t \cup H_2^t \cup P') \land G' \land \bar{B}' \land G))
\]

By Def. 5.9, we can thus derive \( \sigma_t \rightarrow^\omega\tau_t \) for

\[
\tau_t = (l(B_c) \cup B_b; S'; B \land (\tilde{B} \land G') \land \bar{B}' \land G; \bar{T}'_k)^y
\]

for some \( \bar{T}' \) and where \( \text{chr}(S') = l(H_1^t \cup L'' \cup p(H^{p'}) \cup p'' \cup p') \) and \( \tilde{B} = (L = H_1^t \cup H_2^t \cup L') \land (\mathcal{P} \triangleright H_1^t \cup H_2^t \cup P') \). Consequently,

\[
S_t(\tau_t) = [(H_1^t \cup L'' \cup B_c; H^{p'} \cup P''; B \land \tilde{B} \land G' \land \bar{B}' \land G \land B_b; \forall)]
\]

Applying (III.9) and Def. 9.3:1 gives us:

\[
S_t(\tau_t) = [(H_1^t \cup L'' \cup B_c; H^{p'} \cup P''; B \land \tilde{B} \land G' \land \bar{B}' \land G \land B_b; \forall)]
\]

Since \( \mathcal{P} = H^{p'} \cup P'' \), we can apply the matching \( (\mathcal{P} \triangleright H_1^t \cup H_2^t \cup P') \) we find in the guard to get

\[
S_t(\tau_t) = [(H_1^t \cup L'' \cup B_c; H^{p'} \cup H_2^t \cup P''; B \land \tilde{B} \land G' \land \bar{B}' \land G \land B_b; \forall)]
\]

As the variables in \( \bar{y}, \bar{y}' \) are disjoint, we apply (III.8), (III.13), and Def. 9.3:2 to receive:

\[
S_t(\tau_t) = [(H_1^t \cup L'' \cup B_c; H^{p'} \cup H_2^t \cup P''; B' \land G \land B_b; \forall)] = [\tau]
\]

We proceed similarly for ApplyPersistent. Hence, for any \( \sigma_t \in \Sigma_t, \tau \in \Sigma_t \) such that \( S_t(\sigma_t) \rightarrow^\omega[\tau] \), there exists a \( \tau_t \in \Sigma_t \) s.t. \( \tau_t \rightarrow^\omega \tau \) and \( S_t(\tau_t) = [\tau] \).

**Fixed points:** Assume that \( \sigma_t = (L; \mathcal{P}; B; \forall) \) is a fixed point in \( \omega_t \). According to Def. 9.5, one of the following applies: (1) There is no rule \( r \in H_1^t \cup H_2^t \setminus H_2^t \leftrightarrow G \mid B_c, B_b \) in \( \mathcal{P} \) such that \( \sigma_t \equiv (H_1^t \cup H_2^t \cup L'; H_1^t \cup H_2^t \cup P'; G \land G' \land \forall) \). (2) For every such rule that exists, its application violates the non-reflexivity condition, i.e. for a hypothetical follow-up state \( [\tau] \), we have \( [\sigma_t] = [\tau] \).
Now consider a state $\sigma_t$ s.t. $S_t(\sigma_t) = [\sigma_t]$. Hence, it is of the form $\sigma_t = \langle \emptyset; S; B; T \rangle^k_v$ s.t. \chr(S) = $l(L) \cup p(P)$. In case (1), no rules in $[P]$ are applicable to $\sigma_t = \langle \emptyset; l(L) \cup p(P); B; T \rangle^k_v$, except for those of the form $c(t) \setminus c(t) \iff T$. The program will quiesce in a state $\sigma_t' = [\sigma_t]$ after finitely many applications of such rules. In case (2) – assuming a non-pathological CHR program – all possible applications are of the type \textbf{ApplyPersistent} (cf. Lemma 11.9). Consequently, all rules applicable to $\sigma_t$ in $[P]$ are of the form $r' @ l(\bar{H}_1) \cup p(\bar{H}^p) \Rightarrow G | p(B_c) \cup B_0$ or $c(t) \setminus c(t) \iff T$.

For each such rule $r' @ l(\bar{H}_1) \cup p(\bar{H}^p) \Rightarrow G | p(B_c) \cup B_0$, we can tell by $\sigma_t \equiv_1 \tau_t$ that $p(B_c)$ is contained in $p(P)$ and $B_0$ is contained in $B$. Hence, we can apply $r'$ to $\sigma_t$, followed by finitely many applications of rules of the form $c(t) \setminus c(t) \iff T$ to finitely derive a state $\tau_t = \langle \emptyset; S; B'; T' \rangle^k_v$ such that $S_t(\sigma_t) = S_t(\tau_t)$ and $r'$ is not applicable to $\tau_t$. We repeat this for every applicable rule $r'$. After finitely many such sequences of derivation steps, no such rule remains applicable. Thus, we can finally derive a fixed point $\tau_t'$ such that $S_t(\sigma_t) = S_t(\tau_t')$.

It follows by Def. 11.6 that for any goal $G, B$, we have $[A_p(G, B)]_g \subseteq A_p([G, B]_g)$. \hfill \Box

**Example 11.16** (Termination Correspondence). The termination behavior of $\omega_1$ programs encoded in $\omega_1$, via the encoding used to prove Lemma 11.15, changes. Consider a program $P$ consisting only of the rule $a \implies a$ that is clearly terminating in $\omega_1$. Its corresponding encoded program $[P]$ is given below.

\[
\begin{align*}
r_1 & \ @ \ a(l) & \implies & \ a(p) \\
r_2 & \ @ \ a(p) & \implies & \ a(p) \\
r_3 & \ @ \ a(p) \setminus a(p) & \iff & \ T 
\end{align*}
\]

It is an acceptable encoding according to Definition 11.7, and hence, answers are preserved. Nevertheless, there exists the following infinite computation.

\[
\begin{align*}
\sigma &= \langle a(l); \emptyset; T; \emptyset \rangle^0_v \\
\vdash_{r_1^n} & \langle \emptyset; a(l) \# 0; T; \emptyset \rangle^0_v \\
\vdash_{r_1^{n+1}} & \langle a(p); a(l) \# 0; T; (r_1, 0) \rangle^0_v \\
\vdash_{r_3^n} & \langle \emptyset; a(l) \# 0; a(p) \# 1; T; (r_2, 1) \rangle^0_v \\
\vdash_{r_2^n} & \langle a(p); a(l) \# 0, a(p) \# 0; T; (r_1, 0), (r_2, 1) \rangle^0_v \\
\vdash_{r_3^n} & \langle \emptyset; a(l) \# 0, a(p) \# 1; a(p) \# 2; T; (r_2, 1), (r_2, 2) \rangle^0_v \\
\vdash_{r_2^n} & \langle a(p); a(l) \# 0, a(p) \# 1, a(p) \# 2; T; (r_1, 0), (r_2, 1), (r_2, 2) \rangle^0_v \\
\vdash_{r_3^n} & \ldots
\end{align*}
\]

The reason for this difference is found in rules $r_2$ and $r_3$: they enforce set semantics on the constraints, supposedly corresponding to irreflexivity in $\omega_1$. However, the non-determinism of $\omega_1$ seems to hinder proper enforcing of irreflexivity via rules.

**Lemma 11.17** ($\omega_p \not\leftrightarrow \omega_1$). There exists no acceptable encoding of $\omega_p$ into $\omega_1$.

**Proof.** Follows from [Gabbrielli et al., 2009]. Gabbrielli et al. [2009] consider only data sufficient answers, however, as there exists no acceptable encoding of the program given in their proof, the negative result carries over to the generic case of answers. \hfill \Box

**Lemma 11.18** ($\omega_1 \rightarrow \omega_p$). There exists an acceptable encoding of $\omega_1$ into $\omega_p$. 

67
11. Discussion

Proof. We show how to encode any $\omega$-program $P$ in $\omega_p$. For every $n$-ary constraint $c/n$ in $P$, there exists a constraint $c/(n+1)$ in $[P]$. In the following, for a multiset of user-defined $\omega$-constraints $M = \{c_1(t_1), \ldots, c_n(t_n)\}$ let $l(M) := \{c_1(l,t_1), \ldots, c_n(l,t_n)\}$, $p(M) := \{c_1(p,t_1), \ldots, c_n(p,t_n)\}$, and $c(M) := \{c_1(c,t_1), \ldots, c_n(c,t_n)\}$. The encoded program $[P]$ is constructed as follows:

Apply rules 1-3 from the proof of Lemma 11.15, but in rule 2 replace $p(B_c)$ with $c(B_c)$. Assign to each of these rules the constant priority 3. Additionally, add the following rules to $[P]$ for each constraint $c/n$ where $l$ is a sequence of $n$ different variables:

$$1 :: c(p, \bar{t}) \land c(c, \bar{t}) \Rightarrow \top$$
$$2 :: c(c, \bar{t}) \Leftrightarrow c(p, \bar{t})$$

The translation of goals is defined as $[\langle \text{L} \uplus \text{P} \rangle]_g := l(\text{L}') \uplus p(\text{P})$.

**Soundness:** Let $S_l : \Sigma_p \rightarrow \Sigma_l$, $\sigma_p = (l(\text{L}') \uplus p(\text{P}) \uplus c(\text{P}_c) \uplus \text{B}; \text{T})^\nu_k \mapsto (\langle \text{L} \uplus \text{L}'; \text{P} \uplus \text{P}' \uplus \text{P}_c \uplus \text{B}; \text{T}^\nu_k \rangle \mapsto (\text{L} \uplus \text{L}' \uplus \text{P} \uplus \text{P}' \uplus \text{P}_c \uplus \text{B} \land \text{T}^\nu_k \rangle)$ where $\text{ch}(S_l) = l(\text{L}') \uplus p(\text{P}') \uplus c(\text{P}_c)$ and $\tau_p = \tau_p \Rightarrow \top \Rightarrow \tau_p$. In the following, we will show that for all $\sigma_p, \tau_p \in \Sigma_p, \sigma_p \Rightarrow \tau_p \Rightarrow \top \Rightarrow \tau_p$ implies $S_l(\sigma_p) \Rightarrow \tau_p$. The proof is analogous to Lemma 11.15 for the rules of priority 3. As $c(\hat{t}) \Rightarrow c(\hat{t}) \Rightarrow c(\hat{t})$ rules of priority 1 and 2 are invariant to $S_l$.

Now assume $\tau_p$ is a fixed point w.r.t. $[P]$. Analogously to Lemma 11.15, $S_l(\tau_p)$ is a fixed point w.r.t. $P$. The only difference being $c(B_c)$ in the body instead of $p(B_c)$, but rules of priority 1 and 2 would then be applicable to convert $c(B_c)$ into $p(B_c)$ modulo set semantics. Therefore, it follows that $A^\nu_{[P]}([G]_g) \subseteq [A^\nu_{[P]}(G)]_g$.

**Completeness:** Analogously to Lemma 11.15, we have that for any $\sigma_p \in \Sigma_p$, $\tau_p \in \Sigma_l$ such that $S_l(\sigma_p) \Rightarrow \tau_p \Rightarrow [\tau_p]$, there exists a $\sigma_p \in \Sigma_p$ such that $\sigma_p \Rightarrow \tau_p$ and $S_l(\sigma_p) = [\tau_p]$. The only change to the proof is that after applying a rule of the encoded program we also apply all possible **Introduce** and **Solve** transitions, as well as all rule applications with priorities 1 and 2 (all these operations are invariant to $S_l$). Hence, the resulting state $\sigma_p$ contains only identified constraints whose first argument is either $l$ or $p$. All constraints with argument $c$ are either replaced by the corresponding one with argument $p$ by the rule of priority 2, or they are removed, because a corresponding constraint already exists.

Now assume $\sigma_p = (\langle \text{L}; \text{P}; \text{B}; \text{T} \rangle)$ is a fixed point of $P$, then there exists $\sigma_p \in \Sigma_p$ with $S_l(\sigma_p) = [\tau_p]$. There are two possible cases:

1. $[\sigma_p]$ is not applicable to any rule $r \in \mathcal{P}$ (when disregarding irreflexivity)

2. all rule applications would violate irreflexivity

In case 1, $\sigma_p$ clearly is a fixed point as well (otherwise the above soundness result violates the assumption).

Therefore, consider case 2. We assume non-pathological programs, so that, according to Lemma 11.9, **ApplyLinear** never violates irreflexivity. Hence, there exists a rule in $[P]$:

$$3 :: r' \otimes l(\hat{H}^1) \uplus p(\hat{H}^p) \Longrightarrow G \mid c(B_c), B_b$$

In the following, for a set $M$ of constraints let $\#M$ denote the corresponding set of identified constraints. Assume $\sigma_p$ is a fixed point, then $\sigma_p = (\emptyset; \#l(\hat{H}^1) \uplus \#p(\hat{H}^p) \uplus \text{B}; \text{T})^\nu_k$ and $CT = \forall (\text{B} \rightarrow (A \land G))$ with $\sigma_p \Rightarrow r' \Rightarrow \tau_p = (c(B_c) \uplus B_b, \#l(\hat{H}^1) \uplus \#p(\hat{H}^p) \uplus \text{B} \land A; \text{T}^\nu_k)$, where $A := \text{ch}(\#l(\hat{H}^1)) = l(\hat{H}^1) \land \text{ch}(\#p(\hat{H}^p)) = p(\hat{H}^p)$. Applying **Introduce** and **Solve**, we get $\tau_p \Rightarrow (\emptyset; \#l(\hat{H}^1) \uplus \#p(\hat{H}^p) \uplus \text{B} \uplus A; \text{T}^\nu_k)$. 68
The rule \( r' \) corresponds to a rule \( r \) in \( P \) and \( \sigma_1 \) is applicable to \( r \), except for irreflexivity (this follows from soundness). The irreflexivity and Theorem 5 imply \( CT \models B \rightarrow \exists x.(H^p \bowtie H^p \bowtie B_c) \land B \land B_h \). Therefore, \( CT \models (B \land B_h \land A) \rightarrow (H^p \bowtie H^p \bowtie B_c) \). It follows that \( CT \models (B \land B_h \land A) \rightarrow \forall c(t, i) \in c(B_c) . \exists c(p, t') \in p(H^p) . t = t' \).

Therefore, for each \( c(t, i) \in c(B_c) \) we can apply the corresponding rule \( 1 :: c(p, t') \land c(t, i) \Leftrightarrow \top \), as \( CT \models \forall (B \land B_h \land A) \rightarrow \exists x.(\chr(c(p, t')) = c(p, t) \land \chr(c(c(t, i))) = c(c(t, i))) \). Therefore, each constraint in \( c(B_c) \) is removed by rules of priority 1 and we get \( \sigma_p \models H^p = \emptyset ; \#l(H^p) \cup \#p(H^p) \cup S; B; T')^y = \tau_p' \), such that the above rule application is prohibited by \( T' \).

Hence, we can w.l.o.g. choose \( \tau_p' \) as \( \sigma_p \) above and repeat the argument. Therefore, we get a state in which the token store prohibits firing any more propagation rules. As no other rules are applicable either, this state is a fixed point corresponding to \( \sigma_1 \) as well.

Lemma 11.19 \((\omega_l \not\models \omega_e)\). There exists no acceptable encoding of \( \omega_l \) into \( \omega_e \).

Proof. Consider the \( \omega_l \) program \( P = (a \Rightarrow b) \). Since \( A^p_1(a) = \{[\{a; b\}; \top; \emptyset]\} \), an acceptable encoding has to satisfy \( A^p_1([\{a; b\}; \top; \emptyset]) = \{[\{a; b\}; \top; \emptyset]\} \). Therefore, \( [\{a\}; \top; \emptyset] \models \Rightarrow [\{a\}; \top; \emptyset] \) where the result state has to be a final state, which is a contradiction to monotonicity of \( \omega_e \).

Lemma 11.20 \((\omega_e \models \omega_l)\). There exists an acceptable encoding of \( \omega_e \) into \( \omega_l \).

Proof (sketch). Replace propagation rules with simplification rules that contain a copy of the head in their bodies.

11.5 Implementation

In this section, we discuss the implementation of a CHR system that adheres to the operational semantics \( \omega_l \). Currently, no direct implementation exists, so that we only discuss in Section 11.5.1 how it would differ from existing implementations. Afterwards, Section 11.5.2 presents a source-to-source transformation that allows us to simulate \( \omega_l \) execution in \( \omega_e \). The existing implementation of \( \omega_p \) can therefore be used for performing derivations following the operational semantics \( \omega_l \).

11.5.1 Correspondence to Existing Implementations

The overall behavior of \( \omega_l \) is not significantly different from \( \omega_l \) and \( \omega_r \). Clearly, \( \omega_l \) is formally a non-deterministic system like \( \omega_l \), so that a refinement similar to \( \omega_r \) may be considered. However, we consider it an allowed freedom of implementations to perform a fixed rule and constraint selection. Therefore, even without formally refining \( \omega_l \), an implementation may, for example, try rule applications in textual order.

The treatment of built-in and CHR constraints remains almost unchanged. Persistent constraints can be implemented just like linear CHR constraints, when labeling them as persistent. Additionally, an implementation should detect duplicates of persistent constraints and eliminate them.

Slightly more effort is required to adjust the code responsible for matching CHR constraints to rule heads. Firstly, while linear CHR constraints can only be used to match one head constraint, \( \omega_l \) requires that a persistent constraint can match multiple head constraints. Secondly, a case distinction is required to distinguish between ApplyLinear and ApplyPersistent.
rule applications, as the resulting body constraints have to be inserted either linearly or persistently.
Finally, the main difference to existing implementations is the irreflexive transition system used for $\omega_1$. However, in terms of an implementation, the required effort is reduced by the following insight: If the post-transition CHR state is considered as a rule, with all CHR constraints in its head and the conjunction of built-ins as guard, one can simply try to match the pre-transition state to that rule. The necessary code can be reused from the normal matching required for rule applications. If there is such a match, then the CHR constraints are equivalent and the pre-transition built-in store already implies the post-transition built-in store, hence, irreflexivity would be violated.

Additional effort needs to be invested to ensure that persistent constraints are equivalent in the pre- and post-transition states. For example, the rule $a \rightarrow a$ turns a linear $a$-constraint into a persistent $a$-constraint, but the distinction between these two is lost, when only considering an $a$-constraint in a rule head. So a minor adjustment to the matching code is required to ensure that persistent and linear constraints match each other, when testing the irreflexivity condition.

11.5.2 Implementation via Source-to-Source Transformation

In this section, we provide an implementation of the operational semantics $\omega_1$ in the form of a source-to-source transformation. A CHR program $\mathcal{P}$ is transformed into a program $\llbracket \mathcal{P} \rrbracket$ such that $\llbracket \mathcal{P} \rrbracket$’s execution in $\omega_1$ is sound and complete with respect to the execution of $\mathcal{P}$ in $\omega_1$. This transformation is based on Lemma 11.18 and assumes a CHR program $\mathcal{P}$ without pathological rules.

For every $n$-ary constraint $c/n$ in $\mathcal{P}$, there exists a constraint $c/(n+1)$ in $\llbracket \mathcal{P} \rrbracket$. In the following, for a multiset of user-defined $\omega_1$-constraints $M = \{c_1(\bar{t}_1), \ldots, c_n(\bar{t}_n)\}$ let

- $l(M) := \{c_1(l, \bar{t}_1), \ldots, c_n(l, \bar{t}_n)\}$,
- $p(M) := \{c_1(p, \bar{t}_1), \ldots, c_n(p, \bar{t}_n)\}$, and
- $c(M) := \{c_1(c, \bar{t}_1), \ldots, c_n(c, \bar{t}_n)\}$.

The rules of $\llbracket \mathcal{P} \rrbracket$ are constructed via the following source-to-source transformation.

1. For every rule $r \in H_1 \setminus H_2 \Rightarrow G \mid B$ in $\mathcal{P}$, and all multisets $H_1^l, H_1^p, H_2^l, H_2^p$ s.t. $H_1^l \uplus H_1^p = H_1$ and $H_2^l \uplus H_2^p = H_2$ and $H_2^p \neq \emptyset$, the following rule is in $\llbracket \mathcal{P} \rrbracket$:

$$3 :: l(H_1^l) \uplus p(H_1^p) \uplus p(H_2^p) \setminus l(H_2^l) \Rightarrow G \mid l(B_c), B_b$$

2. For every rule $r \in H_1 \setminus H_2 \Rightarrow G \mid B_c, B_b$ in $\mathcal{P}$, and all multisets $H_1^l, H_1^p$ s.t. $H_1^l \uplus H_1^p = H_1$, the following rule is in $\llbracket \mathcal{P} \rrbracket$:

$$3 :: l(H_1^l) \uplus p(H_1^p) \uplus p(H_2) \Rightarrow G \mid c(B_c), B_b$$

3. For every rule $\{c(p, \bar{t}), c(p, \bar{t}')\} \uplus H_1 \setminus H_2 \Rightarrow G \mid B$ in $\llbracket \mathcal{P} \rrbracket$, add also the following rule:

$$3 :: \{c(p, \bar{t})\} \uplus H_1 \setminus H_2 \Rightarrow \bar{t} = \bar{t}' \land G \mid B$$
4. For every user-defined constraint \( c/n \) in \( P \), add the following rules, where \( t \) is a sequence of \( n \) different variables:

\[
\begin{align*}
1 & : c(p, t) \land c(c, t) \iff \top \\
2 & : c(c, t) \iff c(p, t)
\end{align*}
\]

Example 11.21 (Encoding of Transitive Hull). We consider the transitive hull program from Example 9.1:

\[
t \ @ \ e(X, Y), e(Y, Z) \implies e(X, Z)
\]

According to the encoding given above, the program is transformed as follows:

\[
\begin{align*}
3 & : e(l, X, Y), e(l, Y, Z) \implies e(c, X, Z) \\
3 & : e(l, X, Y), e(p, Y, Z) \implies e(c, X, Z) \\
3 & : e(p, X, Y), e(l, Y, Z) \implies e(c, X, Z) \\
3 & : e(p, X, Y), e(p, Y, Z) \implies e(c, X, Z) \\
1 & : e(p, X, Y) \land e(c, X, Y) \iff \top \\
2 & : e(c, X, Y) \iff e(p, X, Y)
\end{align*}
\]

The grouping of the rules above reflects the transformation steps 2, 3, and 4. Transformation step 1 is not productive in this example. The fifth rule above is operationally equivalent to 3 :: \( e(p, X, X) \implies e(c, X, X) \), and hence, is redundant, as the resulting constraint will immediately be removed again by the rule with priority 1. Furthermore, transformation step 3 also adds an additional symmetric version of the fifth rule, which was omitted here, as it is operationally equivalent as well.

Execution of a transformed program in \( \omega_p \) is equivalent to execution of the original program in \( \omega_1 \), as the above is the acceptable encoding used in the proof of Lemma 11.18. As opposed to the acceptable encoding into \( \omega_1 \), this encoding also preserves fixed points, which makes it suitable for our implementation via source-to-source transformation.

Example 11.22 (Example Runs of \( \omega_p \) and \( \omega_1 \) Programs). The following example derivation shows how the translated program terminates with a state that corresponds to the result of an execution of the original program in \( \omega_1 \). For clarity’s and brevity’s sake, we do not show all intermediate states and we do not give the states’ respective token stores explicitly.
12. Related and Future Work

The above computation corresponds to the following execution in $\omega_1$:

$$
\sigma = \left[ (e(A, B), e(B, A); \emptyset; \top; \{A, B\}) \right] \\
\mapsto_1 \left[ (e(A, B), e(B, A); e(A, A); \top; \{A, B\}) \right] \\
\mapsto_1 \left[ (e(A, B), e(B, A); e(A, A), e(B, B); \top; \{A, B\}) \right] \\
\mapsto_1 \left[ (e(A, B), e(B, A); e(A, A), e(B, B), e(A, B); \top; \{A, B\}) \right] \\
\not\mapsto_1
$$

The above example demonstrates that a direct implementation of $\omega_1$ performs less computation steps than its simulation via $\omega_p$. Therefore, the source-to-source transformation is important in that it allows us to directly experiment with $\omega_1$ programs, but ultimately, a direct implementation is more desirable.

12 Related and Future Work

We have already discussed different available operational semantics for CHR in Section 5.2 and seen that they tend to be either analytical or pragmatic, i.e. implementation-oriented. Our operational semantics $\omega$ fills this gap, as it has a strong declarative foundation and is implementable in a terminating fashion.

In particular, all operational semantics that are based on the token-store approach to deal with the trivial non-termination problem (cf. for example [Abdennadher, 1997, Duck et al., 2004]), suffer from incompleteness. In contrast, our approach based on persistent constraint offers a more natural way that better corresponds to the abstract operational semantics $\omega_{va}$.

The set-based operational semantics $\omega_{set}$ given by Sarna-Starosta and Ramakrishnan [2007], on the other hand, avoids the token-store in favor of a low-level change in the implementation. From an analytical point of view this change is undesirable, because it leads to an undetermined number of propagation rule applications. In $\omega_{set}$, execution is similar to $\omega_r$, in that an active constraint is selected and has to find partner constraints for rule applications. Once such a constraint finds no more possible rule applications, it becomes inactive, but can later be awakened again due to changes in the built-in store. This leads to a reactivation, which in turn may lead to another firing of a propagation rule. This overall process is hard to deal with in a declarative way, because it requires a detailed observation of the runtime behavior. Sneyers et al. [2005] already showed that CHR is Turing-complete, even for various subclasses of the language [Sneyers, 2008a, Gabbrielli et al., 2010]. Hence, our operational semantics $\omega$ is Turing-complete as well, due to it being sound and complete with respect to $\omega_{va}$. In order to compare it with the existing operational semantics, we continued along the lines of Gabbrielli et al. [2009] by showing its relative expressivity with respect to $\omega_{va}, \omega_1$, and $\omega_p$. This also yielded a source-to-source transformation that can be used to implement $\omega_1$ via $\omega_p$. For the future, a direct implementation of $\omega_1$ would be preferable though.

The current formulations of $\omega_1$ and $\omega_p$ promise to be beneficial in future CHR research. In the remainder of this thesis, we demonstrate its viability and wherever previous work exists, we can see that $\omega_p$ allows for more succinct and clear formulations. Our axiomatic definition of state equivalence from [Raiser et al., 2009] has already been reused in [Sneyers et al., 2010a]. For the operational semantics $\omega$, it might be possible to include the following additional axiom for state equivalence

$$
\langle L \cup L; L \cup P; \mathbb{P}; \mathbb{B}; V \rangle \equiv_1 \langle L; L \cup P; \mathbb{B}; V \rangle.
$$
Intuitively, this means that a linear constraint is superseded by its persistent variant. We initially decided against this axiom, because it widens the gap to linear-logic, in which a persistent constraint is not equivalent to itself and a linear copy. However, if one accepts this limitation, the above axiom would help to simplify the formulation of the operational semantics significantly. Given this axiom, we can for example ensure that each \texttt{ApplyLinear} transition involves exclusively linear constraints. However, we cannot simply include this axiom, but need to reevaluate our above proofs, especially soundness and completeness of the resulting operational semantics.

Another advantage of such an axiom would be more intuitive results for confluence. Consider the following rule under the current definition of $\omega!$.

\[
\text{a, b} \leftrightarrow \text{c}
\]

Given the initial state $\langle \text{a, b; a}; \top; \emptyset \rangle$ we can apply the rule in two different ways, either with the linear or persistent copy of the $\text{a}$-constraint. This results in the following two states, which are not equivalent without the above axiom, but would be equivalent if it was included.

\[
\langle \text{c}; \text{a}; \top; \emptyset \rangle \not\equiv ! \langle \text{a, c}; \text{a}; \top; \emptyset \rangle
\]

In our algebraic investigation of the merge operator we found it to form a commutative monoid together with the set of equivalence classes of states. This was just a first foray into this line of research, namely the algebraization of CHR. We believe it is possible to profit from the abundance of results available in this field, when further investigating the algebraic structures underlying CHR.

For example, the CHR transition system might be expressible as a category, allowing us to investigate category theoretical constructions for CHR, like pushouts or pullbacks. Similarly, the previously mentioned commutative monoid can be extended to an abelian group, for example via the Grothendieck group construction. This would add inverses of states, or in other words, gives us the algebraic option of subtracting from states. This could provide a formalism that allows us to undo operations, like binding of variables, and hence, extend CHR to a non-committed-choice language, similar to CHR$^\vee$.

On a more pragmatic note, associativity of the merge operator makes it a suitable formalism for investigating parallelism in CHR. If a part of a CHR state allows a computation while being independent from the rest of the state, we can express this as $[\sigma] = [\sigma_0] \circ [\sigma']$. Applying this idea to all independent parallel computations possible in a state, then yields

\[
[\sigma] = [\sigma_0] \circ [\sigma_1] \circ \ldots \circ [\sigma_n] \circ [\sigma'],
\]

which clearly separates $n$ independent parallel computations.
12. Related and Future Work
Chapter IV

Improved Program Analysis Methods

If you want truly to understand something, try to change it.
— Kurt Lewin (1890–1947), Psychologist

In this chapter, we discuss improvements of existing program analysis methods for Constraint Handling Rules. In particular, we extend an idea initially proposed by Duck et al. [2006], which is based on invariants. Section 13 investigates the advantages and disadvantages of applying invariants to program analysis.

Next, we discuss invariant-based confluence in Section 14, which is based on observable confluence, given by Duck et al. [2007]. We then extend the existing program equivalence test by Abdennadher and Frühwirth [1999] in Section 15. We present two orthogonal extensions for it: support for non-terminating, non-confluent programs with a shared interface and support for invariant-based program equivalence. Finally, we discuss related and future work in Section 16.

13 Invariants

Invariants have only recently appeared in CHR program analysis [Lam and Sulzmann, 2006]. In this section, we define them for CHR programs and motivate their beneficial effects on program analysis. In Section 13.1, we present their generic definition and demonstrate an invariant for an exemplary number generator CHR program. Then, Section 13.2 discusses their advantages and disadvantages and demonstrates these for an invariant-based termination proof. Finally, Section 13.3 investigates the formal implications invariants have on standard proof techniques common in the CHR literature.

13.1 Definition

An invariant is a property that holds for a subset of all CHR states, such that if we apply rules to one of these states the invariant still holds for the resulting states.

Definition 13.1 (Invariant). A property \( \mathcal{I} \) is an invariant if and only if for all states \( [\sigma] \) where \( \mathcal{I}([\sigma]) \) holds and all \( [\sigma] \rightarrow^* [\tau] \) we have that \( \mathcal{I}([\tau]) \) holds.

Clearly, in terms of an equivalence-based operational semantics it makes no sense to have invariants that hold for one state, but not for another equivalent state. Therefore, we assume
all invariants in this work to either hold or not hold for all states of an equivalence class of
states, and hence, we take the liberty to write $I(\sigma)$ instead of $I([\sigma])$.

The following example introduces a number generator program that we will refer back to
throughout this chapter. We also define a property $J$ that includes implicit assumptions that
have been made during the development of this program.

**Example 13.2 (Number Generator).** Consider the following program that consists of two
rules and generates numbers.

\[
\begin{align*}
todo(M, \text{gen}(M)) & \iff \top \\
todo(M) \setminus \text{gen}(N) & \iff M \neq N \mid L = N + 1, \text{gen}(L), \text{num}(L)
\end{align*}
\]

Beginning with the initial state

\[
[\sigma] = [(\text{gen}(0), \text{upto}(3); \top; \emptyset)]
\]

the above program derives

\[
[\sigma] \xrightarrow{\ast} [\tau] = [(\text{num}(1), \text{num}(2), \text{num}(3); \top; \emptyset)]
\]

This program makes several implicit assumptions about CHR states:

- there is at most one $\text{upto}/1$ constraint,
- if there is a $\text{upto}/1$ constraint, there is also exactly one $\text{gen}/1$ constraint,
- the arguments of the $\text{upto}/1$ and $\text{gen}/1$ constraints are ground integers, and
- the argument of the $\text{gen}/1$ constraint is less than or equal to the argument of the $\text{upto}/1$
  constraint.

For the remainder of this chapter, let $J$ be a property that holds for a state if and only if all
of the above conditions are satisfied.

### 13.2 Advantages and Disadvantages

The previous example already revealed an important benefit of invariants: they allow us to
restrict program executions, more precisely our analysis of a program’s executions, to states
of a certain structure. It is a common problem that program analysis methods are often
too fine-grained, and hence, fail for states that we deem irrelevant in all practical contexts
(cf. [Sulzmann and Lam, 2008]).

**Example 13.3 (cont.).** Consider again the number generator from Example 13.2. If we start
its execution with an initial state that violates one or more of the implicit assumptions, we
get a different behavior from the desired generation of numbers.

For example, the state \( \langle \text{upto}(5), \text{gen}(N); \top; \emptyset \rangle \) is unable to fire the second rule, as clearly $CT \not\models \forall N.5 \neq N$. So in the best case, our initial state is also a final state. Practically however,
execution with such a state may crash the program, when the comparison $M \neq N$ is performed
on an unbound variable $N$.

We now investigate termination as a typical program analysis, in order to demonstrate the
difference of an invariant-based program analysis.
Definition 13.4 (Termination). A CHR program is terminating if and only if for all states \([\sigma]\) there exists no infinite derivation \([\sigma] \mapsto_e [\sigma'] \mapsto_e [\sigma''] \mapsto_e \ldots\).

Example 13.5 (cont.). Performing any suitable termination analysis reveals that the number generator program is non-terminating. This can be seen from the state \((\text{upto}(0), \text{gen}(1); \top; \emptyset)\), which generates infinitely many \(\text{num}/1\) constraints.

Again, one of our implicit assumptions is violated for this state, and hence, we may ponder the question of whether the number generator program is terminating when only considering the states we intended for its usage.

Let us now formulate termination with respect to an invariant. The underlying idea is straightforward: We only consider derivations that begin with a state, which satisfies the invariant. By Definition 13.1, it is guaranteed that all intermediate states, as well as the result state, also satisfy the invariant.

Definition 13.6 (I-Termination). For an invariant \(I\), we say that a CHR program is \(I\)-terminating if and only if for all states \([\sigma]\), for which \(I([\sigma])\) holds, there exists no infinite derivation \([\sigma] \mapsto_e [\sigma'] \mapsto_e [\sigma''] \mapsto_e \ldots\).

A traditional technique to prove termination is to specify a ranking, i.e., a mapping from states to natural numbers, and prove that each rule application decreases this rank (cf. Section 7.1). For the above example, no such ranking exists in general. However, adapting this technique to \(I\)-termination means that the domain of the ranking is no longer required to be the complete set of all CHR states, but the subset of it that contains all states that satisfy \(I\).

Example 13.7 (cont.). Consider again the above number generator program and assume that \(J\) as defined above is an invariant according to Definition 13.1. Then, we define the following ranking \(r\) that maps states to a natural number:

\[
r([\sigma]) := \begin{cases} 
0 & \text{if no } \text{upto}/1 \text{ in } \sigma \\
M - N + 1 & \text{if } \text{upto}(M), \text{gen}(N) \text{ in } \sigma 
\end{cases}
\]

This ranking is not well-defined on the domain of all CHR states (e.g., \(\text{upto}/1\) arguments may not be numbers, or the second expression may map to negative integers). However, given that its domain are the states satisfying the invariant \(J\), it is well-defined. Furthermore, we easily verify for both rules that for \([\sigma] \mapsto_e [\tau]\) we always have \(r([\sigma]) > r([\tau])\). Therefore, the number generator program is \(J\)-terminating.

We have seen that invariants provide the advantage of adjusting the granularity level for program analysis proofs, such that we can restrict our observations to exactly the practically intended states. However, this comes at two significant costs: Firstly, the above example assumed \(J\) to be an invariant. Clearly, in general we have to formally verify this before proceeding. We leave the proof for \(J\) being an invariant to the reader and instead refer to Section 18.3, which contains an invariant proof of a more complex nature, thus showing that such a proof can indeed be non-trivial.

Secondly, program analysis methods that are founded upon the investigation of certain states may be problematic, because these states might violate the invariant. In particular, this situation occurs for the well-known analysis methods available in the CHR literature for confluence and program equivalence. The following section discusses the effect of invariants on these kinds of program analysis methods and presents ways to compensate for that. Afterwards, Section 14 and Section 15 apply these results in order to develop invariant-based program analysis methods for confluence and program equivalence.
13. Invariants

13.3 Implications for Program Analysis

Program analysis is typically required to argue over an infinite number of states, whilst we can only ever hope to analyze a finite number. Therefore, we focus on states that are minimal in the sense that they contain exactly what is required in order to fire a rule. We refer to these states as rule states and the aim of an analysis is to derive results from these states that also hold for infinitely many similar states.

In this section, we discuss the effect of invariants to program analysis. For this, we first take a different look at rule applications: Based on rule states and the merge operator $\bigcirc$, we temporarily remove unnecessary elements from states. This removal, while readily performed in the literature, is for the first time made explicitly in the form of a state splitting lemma. However, the inclusion of invariants can easily invalidate any analyses made for thusly reduced states. Therefore, we then present an approach that can still allow us to derive useful results that are applicable to infinitely many similar states.

**Definition 13.8 (Rule State).** For a rule $(r @ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b)$ let $\mathcal{V}$ be the variables occurring in $H_1, H_2,$ and $G$. Then the state

$$\langle H_1 \uplus H_2; G; \mathcal{V} \rangle$$

is called the rule state of $r$.

In the literature rule states are sometimes referred to as critical states or minimal states.

By duality, we can also consider the merge operation as a split operation: It splits any state that is applicable to a rule $r$ into the rule state of $r$ merged with the remaining elements of the state. Additionally, we need to bridge information on variables shared between these two states through the global variables. This situation is demonstrated in the following example, before Lemma 13.10 formalizes this splitting operation.

**Example 13.9.** Consider the simple rule $a(X) \Leftrightarrow b(X)$ and its application to the state $\sigma = \langle a(3); \top; \emptyset \rangle$. We can derive the rule state $\langle a(X); \top; \{X\} \rangle$ and due to the equivalence $\sigma \equiv \langle a(X); X = 3; \emptyset \rangle$, we identify $X = 3$ as contextual information that is irrelevant to the actual rule application. However, we cannot simply extract this information into a state of the form $\langle \emptyset; X = 3; \emptyset \rangle$, as it would be equivalent to $\sigma$. At this point, we use the global variable $X$ as a bridge between the rule state and the remaining context state. Hence, we get:

$$[[a(X); \top; \{X\}] \circ \{X\} \circ \{X; X = 3; \{X\}] = [\sigma]$$

Next, the above rule can be applied to the rule state:

$$[[a(X); \top; \{X\}] \rightarrow_e [\{b(X); \top; \{X\}]]$$

Due to monotonicity (cf. Lemma 10.4), we are allowed to extract this derivation and later on merge the context state to the result again:

$$[[a(X); \top; \{X\}] \circ \{X\} \circ \{X; X = 3; \{X\}]$$

$$\rightarrow_e [[b(X); \top; \{X\}] \circ \{X\} \circ \{X; X = 3; \{X\}]$$

$$= [[b(X); X = 3; \emptyset]]$$

$$= [[b(3); \top; \emptyset]]$$
Clearly, this is more complicated than directly applying the rule with $X = 3$ kept in the built-in store of the state. However, as the following lemma shows, it allows us to extract — possibly unknown — context elements within a proof and gives us a rule state to work with. As rule states are directly derived from the syntactical representation of rules we often have enough information about them in order to continue.

Lemma 13.10 (State Splitting with $\diamond V$). Let the state $[\sigma]$ be applicable to a rule $r$. Then $\exists [\delta].[\sigma] = [\sigma_r] \diamond V [\delta]$ with $\sigma_r$ being the rule state of $r$ and $V$ being the global variables of $\sigma_r$.

Proof. As $[\sigma]$ is applicable to $r$, we have by Def. 9.5 that

$$[\sigma] = [(H_1 \uplus H_2 \uplus G; G \wedge B; V')]$$

such that $\sigma_r = (H_1 \uplus H_2; G; V')$ is the rule state of $r$. W.l.o.g. all variables in $\sigma_r$ are fresh and global, such that $V' \cap V = \emptyset$ and that for $\delta ::= (G; B; V \cup V')$ the expression $[\sigma_r] \diamond V [\delta]$ is well-defined. Therefore,

$$[\sigma_r] \diamond V [\delta] = [(H_1 \uplus H_2 \uplus G; G \wedge B; V')] = [\sigma].$$

(The global variables of $\delta$ must include $V$, as variables from $V$ may otherwise occur as local variables in $\delta$, preventing the merge with $\sigma_r$.)

We will see in Sections 14 and 15 that a typical technique in program analysis is to only consider rule states and derive enough information from them in order to argue for all possible states. In an invariant-based analysis, this approach is hindered by the fact that rule states may not satisfy the invariant. In that case, all information gained about such a state becomes irrelevant and the underlying idea, to exploit monotonicity to reason over larger states, fails. Figure IV.1 presents this situation and how we can remedy it using the partial order $\preceq$.

Assume $[\sigma]$ is an investigated state, for which the invariant is not satisfied. Then, we are looking for the states $[\sigma_1], \ldots, [\sigma_k]$, which are minimal elements with respect to $\preceq$, such that $[\sigma \diamond \sigma_i]$ satisfies the invariant. Normally, $[\sigma_0]$ is the least element, however, $[\sigma \diamond \sigma_0] = [\sigma]$ invalidates the invariant.

Given one such minimal element, we can then again apply monotonicity for larger states, i.e. results gained for $[\sigma \diamond \sigma_1]$ can then be applied to $[\sigma \diamond \sigma_1] \diamond V [\sigma']$ for all $\sigma'$, such that the resulting state satisfies the invariant. Again, states $\sigma'$ for which the merge operation results in a state invalidating the invariant need not be considered.
13. Invariants

Definition 13.11 (Minimal Elements). For an invariant $I$, let $\Sigma^I([\sigma]) := \{[\sigma'] \mid I([\sigma] \circ [\sigma'])\} \land \sigma' \text{ has no local variables}\}$. The set $\mathcal{M}^I([\sigma])$ is the set of $\prec$-minimal elements of $\Sigma^I([\sigma])$ such that

$$\forall [\sigma'] \in \Sigma^I([\sigma]). \exists [\sigma_m] \in \mathcal{M}^I([\sigma]). [\sigma_m] \prec [\sigma'].$$ 

Depending on the available built-in constraints and the invariant $I$, the set $\mathcal{M}^I([\sigma])$ may not always be well-defined, or may be infinite. For automated program analyses we would require a finite set, but the approaches presented in this work are generic proof techniques and remain applicable for an infinite set of minimal elements as well.

Example 13.12. Assume the existence of $\prec$-built-in constraints on integer numbers with the usual meaning and let the invariant $I$ hold if and only if for every constraint $n(N)$, the built-in store implies $N < k$ for some constant $k$. Let $\sigma = \langle n(N); \top; \{N\}\rangle$ and let $\sigma(i) = \langle \emptyset; N < i; \{N\}\rangle$ for any integer $i$. Then, the set $\Sigma^I([\sigma])$ contains the infinite series of states $[\sigma(2)], [\sigma(1)], [\sigma(0)], [\sigma(-1)], \ldots$ with $\ldots \prec [\sigma(-1)] \prec [\sigma(0)] \prec [\sigma(1)] \prec [\sigma(2)]$. For none of these states exists a $\prec$-minimal element $[\sigma_m]$ such that $\mathcal{M}^I([\sigma])$ is not well-defined.

The restriction on local variables for $\Sigma^I([\sigma])$ made in Definition 13.11, is justified by two observations: Firstly, consider the states $\sigma_1 = \langle c(X); \top; \emptyset\rangle$ and $\sigma_2 = \langle c(X); X = 1; \emptyset\rangle$. Clearly, we would like to only have to investigate $\sigma_1$, however, we would have to take both of these states, and hence, infinitely many states, into the set $\mathcal{M}^I([\sigma])$, because neither $[\sigma_1] \prec [\sigma_2]$, nor $[\sigma_2] \prec [\sigma_1]$ holds. And secondly, in program analysis we typically consider states with only global variables, because Lemma 10.4 ensures, that all results remain applicable if any of these variables is made local.

The previous discussion of the partial order $\prec$ and minimal elements yields the following proof technique for invariant-based program analysis: In order to prove a property $P$, we perform an induction-like proof. First, we analyze the base case, i.e. for each investigated state we determine its set of minimal elements, which satisfy the invariant. We then prove, that $P$ holds for each investigated state merged with one of its minimal elements. Second, in analogy to an induction step, we show that the property $P$ remains satisfied, when arbitrary states are additionally merged to the states investigated in the base case. Of course, the induction step considers only merging states such that the resulting states satisfy the invariant. An important insight is that all results presented in the following sections can be specialized again: A trivial invariant $I$, that holds for all states, yields the original program analysis methods. In particular, when states are investigated that already satisfy the invariant, we observe that the set of minimal elements gets reduced to the set containing only the least element $[\sigma_0]$ of the partial order $\prec$. We can then assume the set of minimal elements to be defined, even when there are no minimal elements with respect to $\prec$, because the empty state relates to every other state via $\prec$, and hence satisfies the requirement from Definition 13.11.

Lemma 13.13. For an invariant $I$ and a state $\sigma$ such that $I([\sigma])$ holds, we have $\mathcal{M}^I([\sigma]) = \{[\sigma_0]\}$.

Proof. It holds for all $[\sigma']$ that $[\sigma_0] \prec [\sigma']$. By Definition, $[\sigma_0]$ is also a $\prec$-minimal element of $\Sigma^I([\sigma])$. Due to antisymmetry of $\prec$, it is the only $\prec$-minimal element in $\Sigma^I([\sigma])$, hence, $\mathcal{M}^I([\sigma]) = \{[\sigma_0]\}$.
Excursion: Minimal Elements in [Duck et al., 2007]

The idea of minimal elements was originally proposed by Duck et al. [2006] and is found again in [Duck et al., 2007]. We have already discussed the problems involved with extensions and the partial order on states present in these works before. Here, we specifically consider minimal elements, based on the following definition from [Duck et al., 2007].

Definition 13.14. Let $\Sigma_e(\sigma)$ be the set of all valid extensions of some state $\sigma$, and let $\Sigma^I_e(\sigma) = \{ \sigma_e | \sigma_e \in \Sigma_e(\sigma) \land I(\sigma \oplus \sigma_e) \}$ be the set of all valid extensions satisfying the invariant $I$. Finally, let $M^I_e(\sigma)$ be the $\prec_{\sigma}$-minimal elements of $\Sigma^I_e(\sigma)$.

As we have seen before, $\prec_{\sigma}$-minimal elements cannot exist. Therefore, the set $M^I_e(\sigma)$ is never well-defined. For this reason and the problems discovered in previous excursions, we chose to completely reproduce all results from [Duck et al., 2007] in Section 14, based on our definition of minimal elements and the partial order $\triangleleft$.

14 Confluence

Extending traditional confluence analysis with invariants in CHR has already been investigated by Duck et al. [2007] under the name observable confluence. Therefore, our discussion of the topic will concentrate mainly on the differences between our approach and the one presented in [Duck et al., 2007]. Essentially, we derive the same results, but at the same time, eliminate potential problems with the work of Duck et al. [2007].

14.1 Prelimaries

Confluence has been informally introduced in Section 7.2, and as a reminder, is a property that holds, if all derivations for a given goal result in equivalent final states, independently from the applied rules. In this section, we formally introduce the required definitions for confluence and present existing results from [Abdennadher et al., 1999] for deciding confluence of terminating CHR programs.

In this work, we adapted existing definitions to the operational semantics $\omega_e$ based on equivalence classes. For confluence, this has the important effect, that instead of requiring equivalent final states, we can require the same final state, which is already an equivalence class.

The confluence property is defined over the concept of joinability, according to the following definitions.

Definition 14.1 (Joinability). Two states $[\sigma_1]$ and $[\sigma_2]$ are joinable if and only if there exists a state $[\sigma]$ such that $\sigma_1 \rightarrow^*_e [\sigma]$ and $\sigma_2 \rightarrow^*_e [\sigma]$. We denote joinability of $[\sigma_1]$ and $[\sigma_2]$ with $([\sigma_1] \downarrow [\sigma_2])$.

Definition 14.2 (Confluence). A CHR program is confluent if and only if for all states $[\sigma]$, $[\sigma_1],[\sigma_2]$ holds

$$[\sigma] \rightarrow^*_e [\sigma_1] \land [\sigma] \rightarrow^*_e [\sigma_2] \Rightarrow ([\sigma_1] \downarrow [\sigma_2]).$$

The confluence property is also referred to as the diamond property, due the shape of its typical representation, given in Figure II.5.
14. Confluence

14.1.1 Deciding Confluence of Terminating CHR Programs

In this section, we present the existing decision procedure for confluence. Confluence of non-terminating CHR programs is undecidable, but for terminating programs a decision algorithm was given by Abdennadher et al. [1996]. That algorithm is an adaptation of the work originally made for term rewriting systems by Huet [1980]. Here, we reiterate the steps that lead to this algorithm, because we later extend them to confluence modulo an invariant.

In order to decide confluence, we have to limit the number of states that have to be investigated. Instead of considering all derivable states \( \sigma_1 \) and \( \sigma_2 \) in Definition 14.2, we only consider a single rule application. The resulting property is called local confluence.

**Definition 14.3** (Local Confluence). A CHR program is locally confluent if and only if for all states \([\sigma],[\sigma_1],[\sigma_2]\) holds

\[
[\sigma] \rightarrow_\epsilon [\sigma_1] \land [\sigma] \rightarrow_\epsilon [\sigma_2] \Rightarrow ([\sigma_1] \downarrow [\sigma_2]).
\]

Termination ensures us that any locally confluent program is in fact confluent. This is a consequence of the seminal lemma by Newman [1942], which holds for arbitrary rewriting systems.

**Lemma 14.4** (Newman’s Lemma). A terminating reduction system is confluent if and only if it is locally confluent.

Hence, we have now eliminated many possibilities for the states \( \sigma_1 \) and \( \sigma_2 \), because we only have a finite number of possible rule applications for any given state \( \sigma \). The next step towards a decision algorithm is to reduce the possibilities for such a state \( \sigma \). The underlying idea for this is the consideration of so-called overlap states. These states are composed of the elements found in two different rule heads such that they allow both rules to be applicable. Again, the idea originally stems from the work of Huet [1980], but had to be adapted for CHR by Abdennadher et al. [1996]. A direct transfer of the results for term rewriting systems was hindered by CHR’s support for local variables and built-in constraints.

We now introduce the concepts of overlaps and critical pairs. Our formulation here differs from ones found in the literature (e.g., [Abdennadher et al., 1996, Duck et al., 2007, Frühwirth, 2009]), because we adapted them in order to remain consistent with the rest of our work.

**Definition 14.5** (Overlap, Critical Pair). For any two (not necessarily different) rules of a CHR program with renamed apart variables that are of the form

\[
\begin{align*}
\text{(1)} & \quad r_1 \in H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b \\
\text{(2)} & \quad r_2 \in H'_1 \setminus H'_2 \Leftrightarrow G' \mid B'_c, B'_b
\end{align*}
\]

let \( O_1 \subseteq H_1, O_2 \subseteq H_2, O'_1 \subseteq H'_1, O'_2 \subseteq H'_2 \) such that for \( B := ((O_1 \uplus O_2) = (O'_1 \uplus O'_2)) \land G \land G' \) it holds that \( CT \models \exists B \) and \((O_2 \uplus O'_2) \neq \emptyset\), then the state

\[
\sigma = \langle K \uplus K' \uplus R \uplus R' \uplus O_1 \uplus O_2; B; \forall \rangle
\]

where \( \forall \) is the set of all variables occurring in heads and guards of both rules and \( K := H_1 \setminus O_1, K' := H'_1 \setminus O'_1, R := H_2 \setminus O_2, R' := H'_2 \setminus O'_2 \) is called an overlap of \( r_1 \) and \( r_2 \). The pair of states \((\sigma_1, \sigma_2)\) with

\[
\begin{align*}
\sigma_1 & := \langle K \uplus K' \uplus R' \uplus O_1 \uplus B_c; B \land B_b; \forall \rangle \\
\sigma_2 & := \langle K \uplus K' \uplus R \uplus O'_1 \uplus B'_c; B \land B'_b; \forall \rangle
\end{align*}
\]

is called a critical pair of the overlap \( \sigma \).
Intuitively, the overlaps represent minimal critical states for which two (or more) rule applications are possible that could lead to different results. By construction, the overlap $\sigma$ of rules $r_1$ and $r_2$ and its corresponding critical pair $(\sigma_1, \sigma_2)$ satisfy that $[\sigma] \rightarrow_e [\sigma_1]$ and $[\sigma] \rightarrow_e [\sigma_2]$.

**Example 14.6.** Consider the following rule, which is based on a random access machine simulator in CHR given in [Sneyers, 2008a]. Here, a $\text{mem}(P,V)$ constraint denotes that the memory at position $P$ contains the value $V$ and $\text{assign}(P,V')$ denotes that we want to assign a new value $V'$ to the memory at position $P$.

\[
\text{mem}(P,V), \text{assign}(P,V') \Leftrightarrow \text{mem}(P,V')
\]

This example is also a case of an overlap based on a single rule. We can derive the following overlap state from this rule, by overlapping the two $\text{mem}$-constraints.

\[
\sigma = \langle \text{mem}(P,V), \text{assign}(P,V_1), \text{assign}(P',V_2); P = P' \land V = V'; \{P,P',V,V_1,V_2}\rangle
\]

Applying the above rule to this state leads to the following critical pair, which is not joinable, because the final value of the memory at position $P$ will be different. This is not surprising considering that the above rule effectively models destructive assignment.

\[
\langle \text{mem}(P,V_1), \text{assign}(P,V_2); P = P' \land V = V'; \{P,P',V,V_1,V_2}\rangle, \\
\langle \text{mem}(P,V_2), \text{assign}(P,V_1); P = P' \land V = V'; \{P,P',V,V_1,V_2}\rangle
\]

A decision algorithm for confluence is then available via the following theorem from Abdennadher et al. [1996]. It is based on deriving all possible overlaps and critical pairs of a program and examine them for local confluence.

**Theorem 6 (Confluence Test).** Given a terminating CHR program $P$, if all critical pairs between all rules in $P$ are joinable, then $P$ is confluent.

Practically, the confluence test executes each state of a critical pair in the program and then compares the resulting states for equivalence. Although the derivations required in Definition 14.1 are existentially quantified, this execution can be made in any deterministic implementation of CHR. The reason for this lies within the proof of the lemma by Newman [1942], and basically, ensures us that for a locally confluent program any selected derivation path will produce equivalent final states.

### 14.1.2 Implementation

The CHR confluence test given in Theorem 6 is suitable for direct implementation. To the best of our knowledge the following three implementations of confluence checkers for CHR exist.

[Abdennadher et al., 1996] The earliest version, for which the source code is not publicly available. It is severely limited though, as it only supports single-headed simplification rules. Furthermore, it requires modifications to the source code of the program that is to be tested.
14. Confluence

[Frühwirth, 2005] This version has been used to analyze confluence of a parallel union-find implementation. It similarly suffers from the need to rewrite the whole CHR program such that it can be passed to the checker as input. Nevertheless, it was successfully applied in that work to analyze dozens of critical pairs and helped in proving confluence of the union-find implementation.

[Langbein et al., 2010] Finally, this latest version has been built based on our axiomatic definition of state equivalence in Section 8. It is the first confluence checker implementation founded on a proper state equivalence relation, i.e. it is based on Theorem 3. This state equivalence checker is a module that can also be reused for other purposes. Both, the state equivalence and confluence checker are available under the GPL license.

In [Langbein et al., 2010], this checker was compared to the previous two, showing several advantages. Firstly, a proper state equivalence relation is used as its foundation. Secondly, it parses the CHR source code such that no modifications to the checked programs are required. Thirdly, the implementation that computes and filters critical pairs has been compared to the results given in [Frühwirth, 2005] for the union-find program. We have found discrepancies in the number of critical pairs identified by the checkers from [Langbein et al., 2010] and [Frühwirth, 2005]. We manually verified the checker from [Langbein et al., 2010] to consistently produce the correct number of critical pairs. Hence, we consider it the most suitable implementation currently available for confluence checking of CHR programs.

14.2 Invariant-based Confluence

A typical problem of confluence analysis is that the construction of critical pairs does not take into account, whether such states will ever occur in a program’s execution. Clearly, it will occur if a user directly supplies it as input, however, programs are usually developed with an intended usage in mind. The above example showed this problem for termination and the following Section similarly shows it for confluence and defines invariant-based confluence. Then, Section 14.2.2 adapts the existing theorem for the traditional confluence test in order to decide invariant-based confluence.

14.2.1 Definition

Let us first consider the following example, taken from [Duck et al., 2007], to highlight the importance of confluence with respect to an invariant.

**Example 14.7 (Blocks World).** This example stems from agent-oriented programming and describes the behavior of an agent in a blocks world. The following two rules allow an agent to pick up objects.

\[
\begin{align*}
    r_1 & @ \text{get}(X), \text{empty} \iff \text{hold}(X) \\
    r_2 & @ \text{get}(X), \text{hold}(Y) \iff \text{hold}(X), \text{clear}(Y)
\end{align*}
\]

get(X) denotes an action of the agent intended to pick up object X. The constraint empty denotes that the agent is currently not holding anything, whereas hold(X) applies if it is holding object X. Finally, clear(X) represents the fact that X is not held.

This program is non-confluent, as can be seen from observing the overlap state

\[\sigma = \langle \text{get}(X), \text{empty}, \text{hold}(Y); \top; \{X,Y\}\rangle,\]
which leads to the following critical pair:

\[
\langle \text{hold}(X), \text{hold}(Y); \top; \{X, Y\} \rangle, \langle \text{empty}, \text{hold}(X), \text{clear}(Y); \top; \{X, Y\} \rangle
\]

These two states are clearly not joinable, hence, the program is non-confluent.

However, a closer investigation of the overlap state reveals that it represents an agent that holds nothing (because of the \text{empty} constraint) as well as an object \(Y\) (because of the \text{hold}(Y) constraint). This inconsistency is apparent for humans, yet is at the root of the observed non-confluence.

As stated above, we would like to limit our confluence investigations to states that make sense with respect to the program’s intended usage. Therefore, let us define an invariant \(I\) that holds for a state if and only if

- either the agent holds some element \(X\) or holds nothing and
- there is at most one \text{get}(\_\_) operation at one time.

It is easy to see that \(I\) is preserved by both rules and the remainder of this work will provide us the tools required to analyze confluence of the blocks world program with respect to this invariant.

When we have programs with an intended usage in mind, we want any program analysis to ignore states that are not conform with this usage. Therefore, when we consider invariant-based confluence, we no longer want to investigate the program’s behavior for all states. This leads us to the following definition of invariant-based confluence.

**Definition 14.8 (Invariant-based Confluence).** A CHR program \(P\) is confluent with respect to an invariant \(I\) if and only if for all states \([\sigma], [\sigma_1], \text{ and } [\sigma_2] \text{ where } I([\sigma]) \text{ holds we have}

\[
[\sigma] \not\xrightarrow{e}^* [\sigma_1] \lor [\sigma] \not\xrightarrow{e}^* [\sigma_2] \Rightarrow ([\sigma_1] \downarrow [\sigma_2]).
\]

**14.2.2 Deciding Invariant-based Confluence**

The above example showed that an overlap state may not satisfy an invariant. This is the kind of situation we already described in Section 13.3. Our solution to this problem is the approach presented in Figure IV.1: Instead of investigating the overlap state, we instead merge it with all minimal states that help it satisfy the invariant. We then create individual critical pairs for each of the merged states and investigate confluence for them.

Finding a non-joinable critical pair of this form is a counter-example to confluence. What remains to be shown now is that investigation of these critical pairs is sufficient to prove local confluence for all states that satisfy the invariant.

First, we observe that the process of merging additional states to our overlap state has no influence on the joinability of the resulting critical pairs.

**Corollary 14.9 (Joinability).**
Proof. This is a direct consequence of Lemma 10.4.

Our decision theorem again relies on local confluence, this time adapted to invariants.

**Definition 14.10** (Invariant-based Local Confluence). A CHR program is locally confluent with respect to an invariant $I$ if and only if for all states $[\sigma], [\sigma_1], [\sigma_2]$ where $I([\sigma])$ holds, we have

$$[\sigma] \leadsto_e [\sigma_1] \land [\sigma] \leadsto_e [\sigma_2] \Rightarrow ([\sigma_1] \downarrow [\sigma_2]).$$

We can now show that the existing decision theorem for confluence can be adapted to invariant-based confluence by the approach detailed in Figure IV.1.

**Lemma 14.11** (Invariant-based Local Confluence Test). Let $P$ be a CHR program, $I$ an invariant, and let $\mathcal{M}^I([\sigma])$ be well-defined for all overlaps $\sigma$, then:

$P$ is locally confluent with respect to $I$ if and only if for all overlaps $\sigma$ with critical pairs $(\sigma_1, \sigma_2)$ and all $[\sigma_m] \in \mathcal{M}^I([\sigma])$ holds $([\sigma_1] \circ [\sigma_m] \downarrow [\sigma_2] \circ [\sigma_m])$.


“$\impliedby$”: Let $[\sigma], [\sigma_1], [\sigma_2]$ where $I([\sigma])$ holds and $[\sigma] \leadsto_{e_1} [\sigma_1]$ and $[\sigma] \leadsto_{e_2} [\sigma_2]$.

By Def. 14.5, there exists w.l.o.g. an overlap $\sigma_o = \langle \_; \_; \_; \_; \_ \rangle$ of the rules $r_1$ and $r_2$ such that for $[\delta] := ([\langle G; B; V \rangle])$

- $[\sigma] = [\sigma_o] \circ [\delta]
- [\sigma_o] \leadsto_{e_1} [\sigma'_1]$ with $[\sigma_1] = [\sigma'_1] \circ [\delta]$
- $[\sigma_o] \leadsto_{e_2} [\sigma'_2]$ with $[\sigma_2] = [\sigma'_2] \circ [\delta]$

If no such overlap exists, it is because the two rule applications are independent, in which case joinability is trivially true.

As $I([\sigma])$ holds, we have $[\delta] \in \Sigma^L([\sigma_o])$, and therefore, $\exists [\sigma_m] \in \mathcal{M}^I([\sigma_o]), [\sigma_m] < [\delta]$. It follows by definition of $<$ that $\exists [\delta'] : [\delta] = [\sigma_m] \circ [\delta']$, and hence, $[\sigma] = [\sigma_o] \circ [\sigma_m] \circ [\delta']$. By Lemma 10.7 we get $[\sigma] = ([\sigma_o] \circ [\sigma_m]) \circ [\delta']$.

Analogously, we find that $[\sigma_i] = ([\sigma'_i] \circ [\sigma_m]) \circ [\delta']$ for $i = 1, 2$. By the precondition we have $([\sigma'_1] \circ [\sigma_m] \downarrow [\sigma'_2] \circ [\sigma_m])$, and therefore by Corollary 14.9 also, $([\sigma_1] \downarrow [\sigma_2])$.

For invariant-based confluence, we can again determine it for a terminating program by proving local confluence with respect to the invariant.

**Corollary 14.12** (Invariant-based Confluence Test). Let $I$ be an invariant, and $P$ a $I$-terminating CHR program. If $P$ is locally confluent with respect to $I$, then $P$ is confluent with respect to $I$.

Proof. We apply Newman’s lemma (cf., 14.4) analogously to the case without invariants. The applicability of Newman’s lemma follows directly from considering a modified rewriting system, for which the set of states is reduced to only those states that satisfy the invariant.

**Example 14.13** (Blocks World, cont.). As our blocks world example does not require built-in constraints, we know that $\mathcal{M}^0([\sigma])$ is well-defined for all states $\sigma$. Therefore, the prerequisites of Lemma 14.11 are satisfied.
The two rules \( r_1 \) and \( r_2 \) only allow for the single overlap

\[
\sigma = \langle \text{get}(X), \text{empty}, \text{hold}(Y); \top; \{X, Y\} \rangle.
\]

As discussed above, this overlap invalidates the invariant \( B \). However, the inconsistent presence of both the empty and the hold\( (Y) \) constraint cannot be fixed by merging additional constraints. Therefore, \( \Sigma^B(\sigma) = \emptyset \), and in turn, we also have no minimal elements. Nevertheless, Lemma 14.11 is satisfied, because the required condition trivially holds for all minimal elements.

This argument already proves that the blocks world program is locally confluent with respect to \( B \). As the invariant ensures that there is at most one get constraint, the program is also clearly \( B \)-terminating. Therefore, we can apply Corollary 14.12 to determine that the blocks world program is in fact confluent with respect to \( B \).

In the above example, we have seen a case, in which an overlap invalidates the invariant so significantly, that merging another state cannot fix it. This situation is not uncommon, and as we will see later in Chapter V, it can be exploited to eliminate the need for investigating numerous critical pairs.

As mentioned before, the result from Corollary 14.12 has initially been given by Duck et al. [2006]. Our formulation, based on the partial order \( \prec \), fixes potential problems of their approach. Therefore, we refrain from further discussion of invariant-based confluence and its examples, because it would merely be a duplication of the work in [Duck et al., 2006] and [Duck et al., 2007]. Instead we focus on introducing the concept of invariants to program equivalence in the next section and apply invariant-based confluence again in Section V.

15 Program Equivalence

CHR is well-known for its powerful program equivalence test, originally given by Abdennadher and Frühwirth [1999] under the name operational equivalence. It allows us to determine if two given confluent and terminating CHR programs yield equivalent resulting states when given the same input (cf. Section 7.3).

Similarly to the other program analysis methods discussed before, the program equivalence test may fail in practical cases due to investigating irrelevant states. Hence in this section, we extend it to an invariant-based program equivalence test in order to alleviate these problems and make it applicable to a larger set of CHR programs. Furthermore, we apply additional orthogonal ideas to generalize the original algorithm, which also allows us to reason over more CHR programs.

Example 15.1 (Number Generators). Reconsider the number generator program, given in Example 13.2, which we have shown to be terminating only for an invariant \( J \). In general however, it is non-terminating, hence, the program equivalence test given in [Abdennadher and Frühwirth, 1999] is not applicable to it. This could be fixed by restricting the recursion rule to the cases \( M > N \) and replace the first rule by a rule that removes the \( \text{upto}(M) \) and \( \text{gen}(N) \) constraint if \( M \leq N \).

Nevertheless, let us now consider using this number generator in a bigger context, where we add a \( \text{gen}_\text{nums}(N) \) constraint with a ground non-negative integer argument \( N \) and intend
it to generate $\text{num}(n_i)$ constraints, where $n_i = i$ for all $i = 1, \ldots, N$. Hence, we straightforwardly extend our number generator program as follows.

\[
\begin{align*}
gen\_\text{nums}(N) & \Leftrightarrow \text{upto}(N), \text{gen}(0) \\
\text{upto}(M), \text{gen}(M) & \Leftrightarrow \top \\
\text{upto}(M) \setminus \text{gen}(N) & \Leftrightarrow M \neq N \mid L = N + 1, \text{gen}(L), \text{num}(L)
\end{align*}
\]

Clearly, the generation process works from smaller numbers up towards larger numbers. Of course, we could also generate the numbers in the other direction, as done by the following program.

\[
\begin{align*}
gen\_\text{nums}(N) & \Leftrightarrow \text{downfrom}(N), \text{tmp}(N) \\
\text{downfrom}(M), \text{tmp}(0) & \Leftrightarrow \top \\
\text{downfrom}(M) \setminus \text{tmp}(N) & \Leftrightarrow N > 0 \mid \text{num}(N), L = N - 1, \text{tmp}(L)
\end{align*}
\]

These two programs reveal a typical problem: from a practical perspective, we have two different number generators, from which we plan to select one for a larger program. The first rule of each of these programs invokes the corresponding number generator in the intended way. Program equivalence testing should now be able to inform us, whether we may legitimately replace one program by the other, by ensuring us that the ultimately generated numbers in each program remain the same.

While this question may appear intuitively easy, notice that the program equivalence test by Abdennadher and Frühwirth [1999] is already complicated on multiple levels:

- The first program is generally non-terminating. Hence, the preconditions of the test are not satisfied.

- Our assumption that the gen\_nums argument is a ground non-negative integer is ignored. Hence, the input state $(\text{gen\_nums}(N); \top; \{N\})$ only leads to one rule firing in both programs, and therefore, to non-equivalent results.

- The fine-grained operational behavior of these programs differs in terms of the generation order of the numbers. Hence, any analysis that cannot compute all the intended numbers, fails to get equivalent results.

Given our intentions on the usage of these programs, we know that none of the above arguments constitutes a real problem. We do have termination for proper inputs, the rules will fire properly for ground non-negative integers, and in the end our multiset of CHR constraints contains all numbers from 1 to $N$.

Invariant-based program equivalence provides us with an extended notion of program equivalence, which elegantly avoids all of the above problems.

The remainder of this section is structured as follows. First, we recapitulate the results given by Abdennadher and Frühwirth [1999] on operational equivalence in Section 15.1. In particular, their main result is only applicable to terminating and confluent CHR programs. In Section 15.2 we discuss how to avoid these restrictions, before finally, Section 15.3 introduces invariant-based program analysis.
15.1 Preliminaries

In this section, we quickly introduce the results for operational equivalence given by Abdennadher and Frühwirth [1999]. As before, we adjust the presentation to our equivalence-based operational semantics for uniformity. Unless stated otherwise, all definitions and theorems given in this section are from [Abdennadher and Frühwirth, 1999].

The general idea of operational or program equivalence is that two programs should produce equivalent final states for the same input state. For the remainder of this work, we will use \( P_1 \) and \( P_2 \) to denote the two CHR programs to be compared for equivalence. Furthermore, we distinguish derivations made within either of the programs by \( \rightarrow P_1 \) and \( \rightarrow P_2 \), respectively.

The following definition formalizes operational equivalence.

**Definition 15.2 (Operational Equivalence).** A state \( \sigma \) is \( P_1, P_2 \)-joinable if and only if there are two derivations \( [\sigma] \rightarrow P_1 [\tau] \) and \( [\sigma] \rightarrow P_2 [\tau] \) and \( [\tau] \) is a final state with respect to \( P_1 \) and \( P_2 \).

\( P_1 \) and \( P_2 \) are operationally equivalent if and only if all states are \( P_1, P_2 \)-joinable.

The main result of Abdennadher and Frühwirth [1999] is the following sufficient and necessary condition for terminating and confluent programs to be operationally equivalent.

**Theorem 7 (Deciding Operational Equivalence).** Let \( P_1 \) and \( P_2 \) be terminating and confluent CHR programs. \( P_1 \) and \( P_2 \) are operationally equivalent if and only if all rule states of \( P_1 \) and \( P_2 \) are \( P_1, P_2 \)-joinable.

The above theorem is straightforwardly automatable. Rule states can be derived syntactically from the rules and executed individually in each of the programs. As we are guaranteed termination and confluence, we can then directly compare equivalence of the resulting final states.

**Example 15.3 (cont.).** As discussed above, Theorem 7 is not applicable to the programs given in Example 15.1. In fact, we can add another reason, why the two programs are not operationally equivalent according to Theorem 7.

The state \( \sigma = \langle \text{upto}(M), \text{gen}(M); \top; \{M\} \rangle \) is a rule state of the first program, however, using it as input to the second program makes no sense. The second program can handle neither the \( \text{upto} \) nor the \( \text{gen} \) constraint, hence, the state \( \sigma \) is a final state with respect to the second program.

The problem of differing constraint symbols, discussed in the above example, is a general weakness of Theorem 7, or more precisely, of Definition 15.2. It was also identified by Abdennadher and Frühwirth [1999], however the proposed solution was given in the historical context of CHR as a language for writing constraint solvers. In that context, a program’s purpose is to solve a single constraint \( c \), hence, *operational \( c \)-equivalence* was defined. For the interested reader, the following excursion discusses it in more detail. However, for the purpose of CHR as a general-purpose programming language, the notion of operational \( c \)-equivalence is no viable solution. Instead, in Section 15.2 we present our proposed approach based on the *interface* of CHR programs. It is a generalization, as it allows a set of constraint symbols to be shared between programs, instead of only a single constraint symbol.
Excursion: Operational c-Equivalence

Abdennadher and Frühwirth [1999] approached the above problem by assuming that the two CHR programs’ purpose is to solve a certain constraint \( c \). Then, only input states that contain \( c \)-constraints need to be considered. A state, for which the CHR constraints only make use of the constraint symbol \( c \) is called a \( c \)-state, and this leads to the following definition of operational \( c \)-equivalence.

**Definition 15.4** (Operational \( c \)-Equivalence). Let \( c \) be a constraint symbol defined in two CHR programs \( P_1 \) and \( P_2 \). Then, \( P_1 \) and \( P_2 \) are operationally \( c \)-equivalent if and only if all \( c \)-states are \( P_1, P_2 \)-joinable.

Next, a decidable criterion for operational \( c \)-equivalence is required. However, the following example, taken from [Abdennadher and Frühwirth, 1999], reveals that it is not sufficient to only consider \( c \)-states.

**Example 15.5.** Consider the following two CHR programs.

\[
\begin{align*}
    p(a) & \leftrightarrow s & p(a) & \leftrightarrow s \\
    p(b) & \leftrightarrow r & p(b) & \leftrightarrow r \\
    s, r & \leftrightarrow \top
\end{align*}
\]

There are two rule states in these programs, which are \( p \)-states, and indeed, both result in the same final states in both programs. However, the \( p \)-state that contains \( p(a) \) and \( p(b) \) rewrites to the empty state only in the left program.

This dependency of other constraint symbols \((r, s)\) on the fixed constraint symbol \((p)\) has been approximated by Abdennadher and Frühwirth [1999] in the following way.

**Definition 15.6** (Dependent Constraint Symbols). A constraint symbol \( c \) depends directly on a constraint symbol \( c' \), if there is a rule in whose head \( c \) appears and in whose body \( c' \) appears. A constraint symbol \( c \) depends on a constraint symbol \( c' \), if \( c \) depends directly on \( c' \), or if \( c \) depends on a constraint symbol \( d \) and \( d \) depends on \( c' \).

The dependency set of a constraint symbol \( c \) is the set of all constraint symbols that \( c \) depends on. Let \( C_{P_1}, C_{P_2} \) be the dependency sets of \( c \) with respect to \( P_1 \) and \( P_2 \), respectively. Each constraint symbol from \((C_{P_1} \cap C_{P_2}) \cup \{c\}\) is called a \( c \)-dependent constraint symbol.

For their decision theorem, the consideration of rule states that are \( c \)-states was insufficient. Instead, all rule states with only constraint symbols that are \( c \)-dependent constraint symbols are investigated. In [Abdennadher and Frühwirth, 1999] these states are referred to as \( c \)-critical states.

Finally, the below theorem is given as a sufficient criterion for operational \( c \)-equivalence.

**Theorem 8** (Deciding Operational \( c \)-Equivalence). Let \( c \) be a constraint symbol defined in two confluent and terminating CHR programs \( P_1 \) and \( P_2 \). Then the following holds: \( P_1 \) and \( P_2 \) are operationally \( c \)-equivalent if all \( c \)-critical states are \( P_1, P_2 \)-joinable.

Unfortunately, this theorem is not a necessary criterion for operational \( c \)-equivalence. A corresponding counter-example is found in [Abdennadher and Frühwirth, 1999] and the reason they have given for this behavior is the approximating definition of \( c \)-dependent constraint.
symbols. The above definition only derives these constraint symbols from the syntactic structure of rules, but fails to take rule applicability into account.

In the case of only the constraint symbol \( c \) being shared between the two programs, the \( c \)-dependent constraint symbols consist exactly of the constraint symbol \( c \) itself. In that case, the above theorem becomes a necessary and sufficient criterion for operational \( c \)-equivalence.

**Theorem 9** (Necessary Condition for Operational \( c \)-Equivalence). Let \( c \) be the only constraint symbol shared by two confluent and terminating CHR programs \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \). Then the following holds: \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) are operationally \( c \)-equivalent if and only if all \( c \)-critical states are \( \mathcal{P}_1, \mathcal{P}_2 \)-joinable.

### 15.2 Reduced Restrictions for Program Equivalence

After introducing operational equivalence, we now investigate its restrictions and propose ways to reduce them. Our definition of program equivalence, given in Section 15.3, is based on these ideas, which are orthogonal to the introduction of invariants.

#### 15.2.1 Non-Terminating Programs

Theorem 7 requires terminating programs as a prerequisite. Allowing non-termination hinders the automatic decision of program equivalence, as we can no longer execute rule states in the given programs in order to determine the resulting final states. However, we believe that our criterion for program equivalence should be as general as possible.

Hence, we want to be able to apply it in order to decide program equivalence for non-terminating programs, even though that may not be possible automatically. On the other hand, if the programs are known to be terminating, we would like to be able to automate the criterion again.

The proof of Theorem 7 required termination, because it directly considers finite derivations. In order to be able to reuse this proof structure, we instead base our program equivalence definition on finite derivations that lead to final states.

Focusing on final states of a computation, however, is different from requiring termination. For example, two programs that both only contain the rule \( a \Leftrightarrow a \) are non-terminating, but clearly equivalent. In fact, given the input state \( \langle a; \top; \emptyset \rangle \) both programs derive no final states, and therefore, all final states are equivalent.

#### 15.2.2 Non-Confluent Programs

Another prerequisite of Theorem 7 for the two programs is confluence. We assume this restriction has been made with the implementational aspect of the theorem in mind. However, if a program is (intentionally) non-confluent we may still want to compare it with another non-confluent program. In that case, equivalence would have to take all reachable final states into account.

To this end, we will modify Definition 15.2 to consider the set of all possible final states for a state \( \sigma \). This closely corresponds to the definition of \( n \)-equivalence, or normal form equivalence, given by Mohan [1991] for term rewriting systems. In combination with the above arguments on termination, we may then consider programs that are non-confluent and non-terminating and still consider them equivalent as long as their sets of reachable final states for each state are the same.
15. Program Equivalence

**Excursion:** Proof of Theorem 7

Unfortunately, [Abdennadher and Frühwirth, 1999] only contains a simplified proof for Theorem 7 for a subset of all possible CHR programs. Personal communication with the authors revealed that the full proof is no longer available.

Our main theorem, given below, is a generalization of the results given in [Abdennadher and Frühwirth, 1999] (cf. Section 15.3.2). Hence, we can reduce it to theirs, such that our proof doubles as a proof for Theorem 7, more precisely, for its reformulation in the equivalence-based operational semantics.

The fact that our theorem supports non-confluent programs, reveals that the original result of Abdennadher and Frühwirth [1999] also applies to non-confluent programs. This strengthens our assumption that their confluence requirement was introduced to simplify implementations of the theorem.

### 15.2.3 Program Interfaces

The attentive reader will have noticed that in Example 15.1 we have renamed apart all constraint symbols, except gen_nums and num. We could have reused the constraint symbol gen, for example, however this poses a general problem: We want to compare two programs in terms of their output for the same inputs, but what if that input contains constraint symbols only known to one of the programs? In those cases it is clear, that only one program will be able to compute anything, and hence, a program equivalence test is bound to fail.

Abdennadher and Frühwirth [1999] have considered this problem before and proposed to only consider input states that consist of constraints with a certain constraint symbol $c$. This proposal makes sense when considered in the historic setting of using CHR to implement constraint solvers, but in this work we want to be more generic and allow multiple shared constraint symbols. To this end, we introduce the following definition of interfaces of two programs.

**Definition 15.7 (Interface).** Let $P_1, P_2$ be two CHR programs, then their interface is the set $I := \mathcal{C}(P_1) \cap \mathcal{C}(P_2)$.

Intuitively, an interface determines the set of constraint symbols we may use for inputs, as well as those that may be contained in the final resulting states. It assumes programs to be developed with this input-output relation in mind, and hence, all other constraint symbols take the place of intermediate control and data structures that eventually get removed before obtaining the final output.

We are talking about constraint symbols here, although there is technically a distinction between CHR and built-in constraint symbols. However, in the context of comparing two programs we implicitly assume that the underlying constraint theories are the same, hence, both programs are assumed to be able to handle the same set of built-in constraints and we instead focus our attention on the CHR constraint symbols only.

**Example 15.8 (Number Generators, cont.).** The two number generators given in Example 15.1 have the following sets of constraint symbols

\[
\mathcal{C}(P_1) = \{ \text{gen\_nums} / 1, \text{upto} / 1, \text{gen} / 1, \text{num} / 1 \}
\]

\[
\mathcal{C}(P_2) = \{ \text{gen\_nums} / 1, \text{downfrom} / 1, \text{tmp} / 1, \text{num} / 1 \}.
\]
Therefore, their interface is $I = \{ \text{gen\_nums} / 1, \text{num} / 1 \}$, which coincides with our intended usage of the programs: The user specifies via gen\_nums / 1 how many numbers he wants to generate, then the result contains that many num / 1 constraints.

15.3 Invariant-based Program Equivalence

After consideration of the above arguments, let us now define invariant-based program equivalence. It differs from Definition 15.2 by the addition of an invariant $I$ and considering solely states that satisfy it, and instead of joinability compares all resulting states. For the latter, we first define what it means for a state to be $\mathcal{NF}$-equivalent with respect to two programs.

**Definition 15.9 (Normal Forms, $\mathcal{NF}$-Equivalence).** For a state $\sigma$, we call the set

$$\mathcal{NF}_{P}(\sigma) = \mathcal{NF}_{P}([\sigma]) := \{ [\sigma] | [\sigma] \not\mathcal{R}_{P} [\tau] \}$$

the normal forms of $\sigma$.

A state $\sigma$ is $\mathcal{NF}$-equivalent with respect to $P_1$ and $P_2$ if and only if $\mathcal{NF}_{P_1}(\sigma) = \mathcal{NF}_{P_2}(\sigma)$. If $P_1$ and $P_2$ are clear from the context, we simply write $\sigma$ is $\mathcal{NF}$-equivalent. We straightforwardly lift this definition to equivalence classes of states.

Next, we want to restrict our observations to input states that only contain constraint symbols from the interface of the two programs. For an interface $I$, we call such states $I$-states according to the following definition.

**Definition 15.10 (Interface States).** A state $\sigma = \langle G; B; V \rangle$ is called a $I$-state for an interface $I$ if and only if all constraint symbols in $G$ occur in $I$.

We straightforwardly extend this definition to rule states and equivalence classes of states. In the latter case, the equivalence class of failed states is considered a $I$-state for all interfaces $I$.

The same restrictions apply to final or output states, i.e. we assume both programs perform computations on the input that lead to comparable final states without internal constraint symbols. This is covered by the following notion of $\mathcal{NF}$-$I$-equivalence.

**Definition 15.11 ($\mathcal{NF}$-$I$-Equivalence).** Let $\sigma$ be a state and $P_1$ and $P_2$ be CHR programs with interface $I$. Then, $\sigma$ is $\mathcal{NF}$-$I$-equivalent with respect to $P_1$ and $P_2$ if and only if $\sigma$ is $\mathcal{NF}$-equivalent with respect to $P_1$ and $P_2$ and all normal forms in $\mathcal{NF}_{P_1}(\sigma)$ and $\mathcal{NF}_{P_2}(\sigma)$ are $I$-states.

Finally, we can now define our extended program equivalence that includes support for an invariant.

**Definition 15.12 (Invariant-based Program Equivalence).** Let $P_1$ and $P_2$ be CHR programs with interface $I$ and $I$ an invariant. $P_1$ and $P_2$ are $I$-$I$-equivalent if and only if all $I$-states $\sigma$ for which $I([\sigma])$ holds are $\mathcal{NF}$-$I$-equivalent.

The following section adapts Theorem 7 in order to decide invariant-based program equivalence of two programs. Then, Section 15.3.2 discusses different scenarios that may occur in applying the adapted theorem.
15. Program Equivalence

15.3.1 Deciding Invariant-based Program Equivalence

Next, we want to adjust Theorem 7 and make it more general according to the ideas presented in the previous section. Its full proof is no longer available (cf. the above excursion), yet a proof sketch, available in [Abdennadher and Frühwirth, 1999], is based on macro-step induction.

**Definition 15.13 (Macro Step).** Let \( \sigma_r \) be a rule state of \( r \) and let \( [\sigma_r] \rightarrow^+_{\epsilon} [\sigma_f] \) for a final state \([\sigma_f]\). Let \([\sigma]\) be a state applicable to \( r \) and, by Lemma 13.10, let \([\sigma] = [\sigma_r] \circ_{\nu} [\delta] \). A macro step of \([\sigma]\) is a computation of the form \([\sigma] \rightarrow^+_{\epsilon} [\sigma_f] \circ_{\nu} [\delta] \).

Finally, we can now give our main theorem for deciding invariant-based program equivalence. In principle, its proof follows the proof sketch by Abdennadher and Frühwirth [1999], but considers all CHR programs supported by the operational semantics \( \omega_e \). Due to our equivalence-based operational semantics and the corresponding results developed in this work, the full proof requires roughly the same space as the original theorem’s proof sketch, yet covers a significantly larger set of CHR programs.

**Theorem 10 (Deciding Invariant-based Program Equivalence).** Let \( \mathcal{P}_1, \mathcal{P}_2 \) be two CHR programs with interface \( I \) and \( I \) an invariant, for which \( \mathcal{M}^I \) is well-defined for all \( I \)-rule states. \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) are \( I \)-\( \equiv \)-equivalent if and only if for all \( I \)-rule states \( \sigma_r \) of \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) and all \([\sigma_m] \in \mathcal{M}^I([\sigma_r])\) holds that \([\sigma_r] \circ [\sigma_m] \) is \( \mathcal{NF} \)-\( I \)-equivalent.

**Proof.** “\( \Rightarrow \)”:

Let \( \mathcal{P}_1, \mathcal{P}_2 \) be \( I \)-\( \equiv \)-equivalent and let \( \sigma_r \) be a \( I \)-rule state of \( \mathcal{P}_1 \) or \( \mathcal{P}_2 \). If \( I([\sigma_r]) \) holds, then \( \mathcal{M}^I([\sigma_r]) = \{[\sigma_0]\} \) according to Lemma 13.13. By Definition 15.12, \([\sigma_r] \circ [\sigma_0] = [\sigma_r] \) is \( \mathcal{NF} \)-\( I \)-equivalent. Therefore, let \( I(\sigma_r) \) not hold. Then for all \([\sigma_m] \in \mathcal{M}^I([\sigma_r])\) we have that \( I([\sigma_r] \circ [\sigma_m]) \) holds, and hence that \([\sigma_r] \circ [\sigma_m] \) is \( \mathcal{NF} \)-equivalent by precondition.

“\( \Leftarrow \)”:

We have to show for all \( I \)-states \( \sigma \) where \( I([\sigma]) \) holds that \( \mathcal{NF}_{\mathcal{P}_1}(\sigma) = \mathcal{NF}_{\mathcal{P}_2}(\sigma) \) and all normal forms only use constraint symbols from \( I \).

1. \( \mathcal{NF}_{\mathcal{P}_1}(\sigma) \subseteq \mathcal{NF}_{\mathcal{P}_2}(\sigma) \):

Let \([\sigma_f] \in \mathcal{NF}_{\mathcal{P}_1}(\sigma)\), then \([\sigma] \rightarrow_{\mathcal{P}_1}^* [\sigma_f] \) with a finite number of macro steps. We proof by induction over these macro steps that \([\sigma_f] \in \mathcal{NF}_{\mathcal{P}_2}(\sigma) \) and \( \sigma_f \) is a \( I \)-state. In fact, we prove the following slightly stronger hypothesis:

**Induction hypothesis:** For a \( I \)-state \( \sigma' \) with \([\sigma] \rightarrow_{\mathcal{P}_1} [\sigma'] \) and \([\sigma] \rightarrow_{\mathcal{P}_2} [\sigma'] \), we have that \( \sigma' \) is \( \mathcal{NF} \)-\( I \)-equivalent with respect to \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \).

**Base case:** \( \sigma \equiv_{\epsilon} \sigma_f \), then \([\sigma] \in \mathcal{NF}_{\mathcal{P}_2}(\sigma) \) by contradiction:

Let \([\sigma] \rightarrow_{\mathcal{P}_2}^* [\tau] \), by Lemma 13.10 \( \exists [\delta], [\sigma] = [\sigma_r] \circ_{\nu} [\delta] \). As \( I([\sigma]) \) holds, we have \([\delta] \in \Sigma^I([\sigma_r]) \), and hence, \( \exists [\sigma_m] \in \mathcal{M}^I([\sigma_r]), [\sigma_m] \ll [\delta] \). By Definition of \( \ll \) and Lemma 10.7: \([\sigma] = [\sigma_r] \circ_{\nu} ([\sigma_m] \circ [\delta]) = ([\sigma_r] \circ [\sigma_m]) \circ_{\nu} [\delta] \). By precondition, \([\sigma_r] \circ [\sigma_m] \) is \( \mathcal{NF} \)-\( I \)-equivalent and by construction \([\sigma_r \circ [\sigma_m]] \) is not in \( \mathcal{NF}_{\mathcal{P}_2}(\sigma_r \circ [\sigma_m]) \). However, \([\sigma_r \circ [\sigma_m]] \in \mathcal{NF}_{\mathcal{P}_1}(\sigma_r \circ [\sigma_m]) \), which is a contradiction.
Induction Step: By Lemma 13.10, \( \exists [\delta], [\sigma] = [\sigma_r] \circ_V [\delta], \) and analogously above \( \exists [\sigma_m] \in \mathcal{M}^2([\sigma_r], [\sigma_m] \subset [\delta] \) and \( [\sigma] = ([\sigma_r] \circ [\sigma_m]) \circ_V [\delta]. \) Clearly, all the states \( \sigma, \sigma_r, \sigma_m, \sigma, \) and \( \delta \) are \( \mathbb{I} \)-states.

Let \( [\sigma'_J] \in \mathcal{NF}_{P_1}([\sigma_r \circ \sigma_m]), \) then by Lemma 10.4, \( [\sigma] \rightarrow^+_{P_1} [\sigma'_J] \circ_V [\delta]. \) As \( [\sigma_r] \circ [\sigma_m] \) is \( \mathcal{NF} \)-\( \mathbb{I} \)-equivalent by precondition, it follows that \( [\sigma] \rightarrow^+_{P_2} [\sigma'_J] \circ_V [\delta] \) and that \( \sigma'_J \) is a \( \mathbb{I} \)-state. Applying the induction hypothesis to \( [\sigma'_J] \circ_V [\delta] \) we get \( \mathcal{NF}_{P_1}(\sigma) \subseteq \mathcal{NF}_{P_2}(\sigma), \) and that all normal forms of \( P_1 \) are \( \mathbb{I} \)-states.

2. \( \mathcal{NF}_{P_1}(\sigma) \supseteq \mathcal{NF}_{P_2}(\sigma) \) analogously from symmetry of Definition 15.12.

\[ \square \]

15.3.2 Discussion

In this section, we discuss different exemplary scenarios for the application of Theorem 10, beginning with our number generator example.

Example 15.14 (Number Generators, cont.). For the last time, let us reconsider the number generator programs.

\[
\begin{align*}
gen\_nums(N) & \iff \text{upto}(N), \text{gen}(0) \\
\text{upto}(M), \text{gen}(M) & \iff \top \\
\text{upto}(M) \setminus \text{gen}(N) & \iff M \neq N \mid L = N + 1, \text{gen}(L), \text{num}(L) \\
\text{gen}\_nums(N) & \iff \text{downfrom}(N), \text{tmp}(N) \\
\text{downfrom}(M), \text{tmp}(0) & \iff \top \\
\text{downfrom}(M) \setminus \text{tmp}(N) & \iff N > 0 \mid \text{num}(N), L = N - 1, \text{tmp}(L)
\end{align*}
\]

We have seen in previous examples, that an invariant \( \mathcal{I} \) is required to consider the implicit assumptions made for the first program. Let us adapt this invariant to hold for both of the above programs. \( \mathcal{J} \) then requires the following properties to hold for a state:

- there is at most one gen\_nums/1 constraint,
- the argument of any gen\_nums/1 constraint is a ground non-negative natural number.

Furthermore, we have identified their interface \( \mathcal{I} = \{ \text{gen}\_nums/1, \text{num}/1 \}, \) and hence, there exists only the 1-rule state \( \sigma_r = (\text{gen}\_nums(N); \top; \{N\}). \)

We realize, that \( \mathcal{J}([\sigma_r]) \) is invalidated, because the argument is not a ground non-negative natural number. As we only require built-in equivalence in order to supply a corresponding number for \( N \), the set of minimal elements for \( [\sigma_r] \) is well-defined as \( \mathcal{M}^\mathcal{J}([\sigma_r]) = \{ [0; N = k; \{N\}] \mid k \in \mathbb{N} \} \). It is also easy to see, that \( \mathcal{J} \) is indeed an invariant with respect to both programs.

Therefore, all prerequisites of Theorem 10 are satisfied and we can apply it in order to determine, whether the two programs are \( \mathcal{J} \)-\( \mathbb{I} \)-equivalent. Hence, we have to show that for all \( [\sigma_m] \in \mathcal{M}^\mathcal{J}([\sigma_r]) \) holds that \( [\sigma_r] \circ [\sigma_m] \) is \( \mathcal{NF} \)-\( \mathbb{I} \)-equivalent. This scenario demonstrates that, despite applying Theorem 10, we may still need to invest manual effort. Nevertheless, instead of having to investigate all possible CHR states, our theorem allows us to restrict this effort to the set of states \( \{ [\text{gen}\_nums(N); N = k; \{N\}] \mid k \in \mathbb{N} \} \). Therefore, we essentially have to manually prove \( \mathcal{NF} \)-\( \mathbb{I} \)-equivalence for all these relevant input states.
To this end, we perform two traditional induction proofs, showing that both programs reach the same normal forms for the above set of states. For the first program we consider the following property as the induction hypothesis.

Given a state of the form \[\langle \text{upto}(N), \text{gen}(0); N = k; \{N\}\rangle\] for some \(k \in \mathbb{N}\) and a \(l \in \mathbb{N}\) with \(l \leq k\), there exists exactly one possible derivation, which leads to a state of the form \[\langle \text{upto}(N), \text{gen}(l), \text{num}(1), \ldots, \text{num}(l); N = k; \{N\}\rangle\] and for \(l = k\) this state can be applied to the first rule to yield the normal form \[\langle \text{num}(1), \ldots, \text{num}(N); N = k; \{N\}\rangle\].

We begin the induction with the base case, i.e. for \(k = 0\) and the state \[\langle \text{gen} _\text{num}(N); N = 0; \{N\}\rangle\]. In this case, the only choice for \(l \leq k\) is \(l = 0\) and we have

\[\begin{align*}
\Rightarrow_{p_1} & \quad \langle \text{gen} _\text{num}(N); N = 0; \{N\}\rangle \\
\Rightarrow_{p_1} & \quad \langle \text{upto}(N), \text{gen}(0); N = 0; \{N\}\rangle \\
\Rightarrow_{p_1} & \quad \langle \emptyset; N = 0; \{N\}\rangle.
\end{align*}\]

We see that the induction hypothesis is satisfied: The first rule application brings us to a state of the desired form, which is already the required state with \(\text{num}(1), \ldots, \text{num}(l)\) being the emptyset.

For the induction step let us consider the state \[\langle \text{gen} _\text{num}(N); N = n + 1; \{N\}\rangle\] and its first and only possible rule application, which leads to \[\langle \text{upto}(N), \text{gen}(0); N = n + 1; \{N\}\rangle\]. Next, we apply the induction hypothesis for \(l = n\), such that we get a unique derivation from this state, leading to the state \[\langle \text{upto}(N), \text{gen}(n), \text{num}(1), \ldots, \text{num}(n); N = n + 1; \{N\}\rangle\]. In this state, there is exactly one possible rule application again, which leads to

\[\langle \text{upto}(N), \text{gen}(N), \text{num}(1), \ldots, \text{num}(N); N = n + 1; \{N\}\rangle\].

Hence, the first part of the induction hypothesis also holds for \(n + 1\). Furthermore, the above state leads to one final possible rule application, which gives us the desired normal form

\[\langle \text{num}(1), \ldots, \text{num}(N); N = n + 1; \{N\}\rangle\].

Analogously, we verify for the second program, that each of the initial states results in the same unique normal form, hence, proving their \(\mathcal{NF}\) equivalence.

The application of Theorem 10 still required manual labor, however, it gave us an important advantage: Through the set of minimal elements, we immediately got the relevant states and we know that their investigation is sufficient. Compared to only using Definition 15.12, this yields a clearer approach towards the desired goal of proving program equivalence and additionally reduces the required proof effort by limiting it to a much smaller number of relevant states.

**Well-defined minimal elements** Theorem 10 requires the set of minimal elements \(\mathcal{M}^I\) to be well-defined. As discussed above, this may not always be the case. However, we only require it to be well-defined for \(I\)-rule states, which can make a significant difference. We have seen in Example 13.12 that supporting built-in \(<\) and \(>\)-constraints are problematic.

In the above example, a \(>\)-constraint occurs in a guard. Nevertheless, we have well-defined minimal elements for the \(I\)-rule states, because the invariant ensures us ground arguments, hence, \(>\)-constraints can only contain redundant information. We will see a similar case in Chapter V, but in general, this well-definedness may hinder the application of Theorem 10.
**Implementation** Furthermore, the above example showed that there may be infinitely many minimal elements for a \( \mathcal{I} \)-rule state. Clearly, this complicates an implementation of Theorem 10. In this work, we focus on applying the theorem in proofs, but will shortly discuss the requirements for an implementation here.

Firstly, we would like an implementation to terminate. Testing \( \mathcal{N}_F \mathcal{I} \)-equivalence is usually realized by executing the corresponding state in both programs and comparing the resulting final states. This implicitly requires both programs to be terminating. As opposed to [Abdennadher and Frühwirth, 1999], the confluence requirement can be avoided if one is able to compute all possible non-deterministic results. This can, for example, be achieved by results like [Martin et al., 2009] or specialized CHR implementations that support computing all normal forms of a state.

Furthermore, one should take care that the interface \( \mathcal{I} \) is the intended interface. This may require renaming of constraint symbols to ensure that no constraint symbols are shared between the programs, except for those used in input and result states.

While the calculation of \( \mathcal{I} \)-rule states of two programs is straightforwardly automatable, the computation of minimal elements for these is not. Therefore, an implementation of our theorem needs to be adjusted for each invariant with additional code that computes the minimal elements. Again, we require that this code is terminating, i.e. for every \( \mathcal{I} \)-rule state only a finite number of minimal elements may exist.

**Comparison with [Abdennadher and Frühwirth, 1999]** As mentioned above, our theorem is a generalization of the one given by Abdennadher and Frühwirth [1999]. In order to retrieve the original theorem, we make the following instantiations:

- Instead of arbitrary programs \( P_1 \) and \( P_2 \) only terminating and confluent programs are allowed. Hence, \( \mathcal{N}_F \mathcal{I} \)-equivalence compares the two unique resulting final states only.

- All constraint symbols are shared between both programs, i.e. \( \mathcal{I} \) contains all constraint symbols from both programs.

- The invariant \( \mathcal{I} \) holds for all states. Hence, the set of minimal elements always consists only of the empty state.

The resulting instance of Theorem 10 then directly resembles Theorem 7. The above instantiations further demonstrate the points in which our theorem is more powerful: support for non-confluent, non-terminating programs, consideration of interfaces and invariants.

---

**Excursion:** Comparison with Operational \( c \)-Equivalence

In a previous excursion, we have introduced the notion of operational \( c \)-equivalence, as given by Abdennadher and Frühwirth [1999]. In order to reproduce Theorem 8, we need to make similar instantiations as the above, with one exception: The interface \( \mathcal{I} \) should be limited to the \( c \)-dependent constraint symbols.

As with the original results in [Abdennadher and Frühwirth, 1999], a syntactical approximation of the \( c \)-dependent constraint symbols leads to a sufficient, but not necessary, criterion. Similarly, if \( c \) is the only constraint symbol shared by the two programs, i.e. \( \mathcal{I} = \{c\} \), then the theorem becomes a necessary criterion and we have reproduced Theorem 9.
16 Related and Future Work

There still exist programs, for which we know that they operate equivalently in all relevant cases, but for which Theorem 10 fails. A whole class of such programs is represented by the union-find implementation [Schrijvers and Frühwirth, 2006] in CHR. From our input-output perspective, it makes no difference whether a union-find implementation performs union-by-rank and path compression, or not, as long as all elements are assigned their correct unions. However, the different resulting graphs cause the final states to become non-equivalent.

The common denominator of this class of programs is a graph-based data structure, for which different (non-isomorphic) graphs represent equivalent information. This kind of equivalence is beyond the state equivalence relation $\equiv_e$. One might think of an extension of $\equiv_e$ as a solution to this problem, however, this has to be done carefully: The extension has to be compliant with rule application in order to allow usage of the equivalence-based operational semantics $\omega_e$. This compliance may even have to be shown for the specific rules of the program.

In other words, it is unclear how we can extend Theorem 10 in order to handle such programs with it. Clearly, it comes as no surprise that some programs are difficult to handle, because after all, program equivalence itself is undecidable.

In this work, we focused on the input-output relationship of programs for their equivalence. When considering concurrent and non-terminating processes, bisimilarity is usually considered instead. However, it depends on a labeled transition system and it is not evident, how CHR can be formulated as a labeled transition system, such that bisimilarity can be investigated for it.

Fortunately, Leifer and Milner [2000] gave a generic construction for transforming an unlabeled transition system into a labeled transition system. It generates labels that represent a sufficiently large context under which transitions can be compared meaningfully with bisimilarity. It remains to be investigated whether this approach is applicable to CHR, which would lead to a direct applicability of all bisimilarity results to CHR.

Finally, in this work we intentionally used invariants to capture implicit assumptions made during program development. We also assumed that correctness of said invariants is given. Continuing this line of research one might want to consider research on invariants in general. In particular, automatic verification of invariants allows us to more easily apply the results of this section. One might even step into the field of invariant generation in order to answer questions of the form: What is the least invariant $I$ one has to impose on states such that a given program becomes $I$-confluent?
Chapter V

Embedding Graph Transformation Systems in Constraint Handling Rules

Elegance is not a dispensable luxury but a quality that decides between success and failure.

— Edsger Wybe Dijkstra (1930–2002), Computer Scientist

In the introduction, we presented the problems encountered when first embedding graph transformation systems in CHR. These problems have been thoroughly addressed in Chapter III and Chapter IV, so that we can now revisit the GTS encoding.

We begin in Section 17 by presenting the encoding itself, which is rather intuitive thanks to both systems being non-deterministic and rule-based.

We proceed by showing soundness and completeness of this encoding in Section 18. For this purpose, we rely on our equivalence-based operational semantics, given in Section 8. Persistent constraints will not be used, simply because they are a generalization of $\omega_e$ and neither persistent constraints, nor propagation rules, have a direct correspondence in graph transformation systems.

We define an invariant that specifies for CHR states, whether they are valid encodings of a graph. Based on this invariant, Sections 19 and 20 then compare confluence and program equivalence for CHR and GTS.

Although chronologically, our GTS encoding was at the beginning of this work, this chapter is now the culmination of our results given above. We essentially present the results from [Raiser, 2007], yet adapt them to the notations and concepts we introduced earlier in this work. Hence, we can also provide insight into the effects of choosing $\omega_e$ over previous formulations.

17 Encoding of Graph Transformation Systems

In this work, we assume that the CHR programs resulting from encoding a GTS are executed only with encodings of graphs. Naturally, we may provide the CHR programs with completely different inputs, especially with inconsistently encoded graphs. It is clear, that we cannot expect any meaningful results from such computations, hence, for the remainder of this work we restrict all observations to programs and states that correspond to GTSs and graphs. We formalize this restriction in Section 18.1 by means of an invariant.

Therefore, on one hand, any state that violates the invariant will not be considered as input, and on the other hand, any graph can be encoded into a state that satisfies the invariant.
17. Encoding of Graph Transformation Systems

We show in Section 18.3 that execution of the encoded GTS in CHR for invariant-satisfying states always leads to results that also satisfy the invariant. In other words, when providing a graph as input to the CHR program, the result will also be a graph, as is to be expected. For our encoding, we first determine the necessary constraint symbols required for rule and host graphs. At this point, we require the GTS to be typed, so we can directly infer these constraint symbols from the corresponding type graph, as explained in Definition 17.1. This is not a restriction though, as every untyped graph can be considered typed over the trivial type graph (cf. Figure II.4).

Definition 17.1 (Type Graph Encoding). For a type graph $\mathcal{T}G$, we define the set $\mathcal{C}$ of required constraint symbols to encode graphs typed over $\mathcal{T}G$ as the minimal set satisfying:

- If $v \in V_{\mathcal{T}G}$ then $v/2 \in \mathcal{C}$.
- If $e \in E_{\mathcal{T}G}$ then $e/3 \in \mathcal{C}$.

We assume that all constraints introduced by Definition 17.1 have unique names. We begin in Section 17.1 with encoding single graphs, before we encode complete rules in Section 17.2.

17.1 Encoding Graphs

The following definition presents the encoding of a typed graph. The encoding is parametrized such that we have two variants: ground-encodings apply to host graphs, which are supplied as input, whereas keep-encodings do not specify degrees for nodes. We mainly apply keep-encodings for rule encodings, but also discuss its usage for specifying input graphs in Section 18.2.

Definition 17.2 (Typed Graph Encoding). We define the following helpful mappings for an infinite set of variables VARS:

- $\text{type}_G(x)$ denotes the corresponding constraint symbol for encoding a node or edge of the given type.
- $\text{var} : G \rightarrow \text{VARS}, x \mapsto X_x$ such that $X_x$ is a unique variable associated to $x$, i.e. $\text{var}$ is injective for $X$ being the set of all graph nodes and edges. In particular, different graphs result in different variables.
- $\text{dvar} : G \rightarrow \text{VARS}, x \mapsto X_x$ such that $X_x$ is a unique variable associated to $x$, i.e. $\text{dvar}$ is injective for $X$ being the set of all graph nodes and edges and different from $\text{var}$. In particular, different graphs result in different variables.

Using these mappings we define the following encoding of graphs:

$$\text{chr}_G(E, x) = \begin{cases} 
\text{type}_G(x)(\text{var}(x), \text{deg}_G(x)) & \text{if } x \in V_G \land E = \text{ground} \\
\text{type}_G(x)(\text{var}(x), \text{dvar}(x)) & \text{if } x \in V_G \land E = \text{keep} \\
\text{type}_G(x)(\text{var}(x), \text{var}(\text{src}(x)), \text{var}(\text{tgt}(x))) & \text{if } x \in E_G 
\end{cases}$$

We use the notations $\text{chr}(\text{ground}, G) := \{\text{chr}(\text{ground}, x) \mid x \in G\}$ as well as $\text{chr}(\text{keep}, G) := \{\text{chr}(\text{keep}, x) \mid x \in G\}$. Furthermore, we omit the index $G$ if the context is clear. We call $\text{dvar}(v)$ the degree variable for a node $v$.

A host graph $G$ is encoded in CHR as $(\text{chr}(\text{ground}, G); \top; \forall)$, where $\forall$ can be chosen freely.
Example 17.3. For our example of the GTS for recognizing cyclic lists (cf. Example 6.6) we assume the trivial type graph (cf. Figure II.4). Based on this type graph we need the constraints node/2 and edge/3. The host graph $G$, given in Figure V.1, that contains a cyclic list consisting of exactly two nodes is encoded in $\text{chr(ground, } G\text{)}$ as:

$$\begin{align*}
\text{node}(N_1, 2), \text{node}(N_2, 2), \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1)
\end{align*}$$

The same graph $G$ encoded in $\text{chr(keep, } G\text{)}$ has the following form:

$$\begin{align*}
\text{node}(N_1, D_1), \text{node}(N_2, D_2), \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1)
\end{align*}$$

17.2 Encoding Rules

We can now encode a complete graph production rule based on the previous definitions.

Definition 17.4 (GTS Rule in CHR). For a graph production rule $\rho = (L \xleftarrow{K} R \xrightarrow{R})$ from a GTS we define $\varphi(\rho) = (p \circ C_L \Leftrightarrow C_u_R, C_b_R)$ with

- $C_L = \{\text{chr}_{L}(\text{keep}, x) \mid x \in K\} \cup \{\text{chr}_L(\text{ground}, x) \mid x \in L \setminus K\}$
- $C_u_R = \{\text{chr}_R(\text{ground}, x) \mid x \in R \setminus K\} \cup \{\text{chr}_R(\text{keep}, e) \mid e \in E_K\} \cup \{\text{chr}_R(\text{keep}, v') \mid v \in V_K\}$
- $C_b_R = \{\text{var}(v) = \text{var}(v') \land \text{dvar}(v') = \text{dvar}(v) - \deg_{L}(v) + \deg_{R}(v) \mid v \in V_K\}$

Here, we consider nodes and edges that are in $K$ as well as in $L$ or $R$ to be the same (by inclusions $l$ and $r$), such that the corresponding variables given by $\text{var}$ and $\text{dvar}$ are also the same.

A CHR program that is created from a GTS according to the above definition, will be referred to as a GTS-CHR program for the remainder of this work.

Example 17.5 (cont.). As an example, consider the second rule from Example 6.6, which reduces two cyclic nodes to a single node with a loop. Its encoding as a CHR simplification rule is given below:

$$\begin{align*}
\text{twoloop @} & \quad \text{node}(N_1, D_1), \text{node}(N_2, 2), \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1) \\
\quad \Leftrightarrow & \quad \text{node}(N'_1, D'_1), \text{edge}(E_3, N_1, N_1), N'_1 = N_1, D'_1 = D_1 - 2 + 2
\end{align*}$$

It is also possible to simplify the resulting rules, as explained later in Section 17.2.1.

When applying a GTS rule, the gluing condition, according to Definition 6.5, has to be satisfied. Due to our restriction to injective match morphisms, the gluing condition is violated if there exists $x \in DP$ with $x \not\in GP$. Intuitively, when a node gets deleted by a rule, the corresponding node in the host graph may have an edge adjacent to it, which is not explicitly
17. Encoding of Graph Transformation Systems

Figure V.2: Graph with a dangling edge if node 2 is removed by the twoloop rule

given in the rule. In such a case, the remaining edge would be left dangling as it is no longer adjacent to two nodes. Therefore, this situation has to be avoided and before a rule is applied to a host graph, we first have to ensure that there are no dangling edges according to the following definition.

**Definition 17.6** (Dangling Edge). A dangling edge is an edge $e \in E_G \setminus m(E_L)$ such that there is a node $v \in V_L \setminus V_K$ with $m(v) = \text{src}(e) \lor m(v) = \text{tgt}(e)$.

**Example 17.7** (cont.). Consider the twoloop rule given in Example 17.5, along with the following encoded host graph shown in Figure V.2:

$$
\text{node}(V_1, 2), \text{node}(V_2, 3), \text{node}(V_3, 1), \\
\text{edge}(E_1, V_1, V_2), \text{edge}(E_2, V_2, V_1), \text{edge}(E_3, V_2, V_3)
$$

Applying the twoloop rule to this graph to remove the node $V_2$ would leave the edge $E_3$ dangling. However, this is avoided as the encoding of the twoloop rule contains the constraint $\text{node}(N_2, 2)$ in its head. Hence, only a node with a degree of exactly 2 can be removed by this rule. Nevertheless, the rule can be applied with $N_2 = V_1$ as the node $V_1$ has the required degree of 2.

17.2.1 Encoding Alternatives

The encoding presented above can be varied in several different ways. We chose the encoding in Definition 17.2 and Definition 17.4 for this work, because it is verbose, hence, directly presenting all its components and simplifying the proofs. In practice however, a less verbose encoding resulting in shorter rules can be used instead. In this section we present different possible simplifications achieving this.

The different simplifications are illustrated by applying them to the twoloop rule which is of the following form when encoded as specified in Definition 17.4:

$$
twoloop @ node(N_1, D_1), node(N_2, 2), edge(E_1, N_1, N_2), edge(E_2, N_2, N_1)
$$

$$
\Leftrightarrow
\text{node}(N'_1, D'_1), edge(E_3, N_1, N_1), N'_1 = N_1, D'_1 = D_1 - 2 + 2
$$

There are two ways to specify the degree of a node in $L \setminus K$. The one chosen in Definition 17.4 explicitly specifies the respective degree in the head. Another way is to keep the degree as a variable $D$ in the head and add the built-in constraint $D = k$ to the guard of the rule. However, most current CHR compilers detect these equalities and automatically transform between them to get the representation most suitable for optimization. Therefore, in this work we directly specify the degree in the head constraints, in order to avoid guards altogether.

**Variable Elimination** As Definition 17.4 encodes a node $v \in V_K$ using a new node identifier $v'$ with $\text{var}(v) = \text{var}(v')$ and $\text{var}(v')$ is not used elsewhere, this substitution can be included directly into the rule encoding:
We perform variable elimination on node identifiers by default in the remainder of this work. However, as we need to take degree adjustments into account, the formulation of Definition 17.4 is simplified by this variable duplication.

**Arithmetic Simplification**  The degree adjustments in Definition 17.4 explicitly contain the information on how many edges the rule deletes and creates. For the adjustment itself, however, it is sufficient to simply modify the degree by the actual change in the number of edges. Additionally, if the change is 0, like in the `twoloop` rule, the extra local variable used for the degree can be substituted, resulting in:

\[
\text{twoloop } @ \text{ node}(N_1, D_1), \text{ node}(N_2, 2), \text{ edge}(E_1, N_1, N_2), \text{ edge}(E_2, N_2, N_1) \iff \text{ node}(N_1, D_1'), \text{ edge}(E_3, N_1, N_1), D_1' = D_1 - 2 + 2}
\]

**Elimination of Edge Identifiers**  Edge identifier variables are used throughout this work, because they simplify dealing with the multiset semantics of CHR with respect to the edge constraint representing exactly one edge of a graph. In a CHR implementation, however, every constraint is implemented as a unique object – in some operational semantics, like $\omega_t$, even annotated with an identifier number – which makes the explicit edge identifiers redundant. Using this idea the `twoloop` rule can be further simplified to:

\[
\text{twoloop } @ \text{ node}(N_1, D_1), \text{ node}(N_2, 2), \text{ edge}(N_1, N_2), \text{ edge}(N_2, N_1) \iff \text{ node}(N_1, D_1), \text{ edge}(N_1, N_1)
\]

This argumentation cannot be applied to node identifiers however, as those are required for specifying the source and target of edge constraints.

**Simpagation Rules**  Some nodes and edges of the left-hand rule graph $L$ of a GTS rule may occur only to specify a certain graph context and are unaffected by the rule application. This can also happen for nodes, if the modification to adjacent edges results in no change to the degree, as in the `twoloop` rule. In those cases, the node or edge is encoded in exactly the same way in head and body of the rule. Therefore, during the rule application the corresponding constraint is removed and introduced again. A simpagation rule allows us to move such a constraint into the part of the head which is not removed during the rule application. This reduces the textual size of the rule as well as its execution time, because it avoids deletion and regeneration of a constraint during rule application.

A CHR compiler will not make this optimization by default, because in operational semantics, like $\omega_t$ or $\omega_r$, these two alternatives may (intentionally) result in different executional behavior. Due to the regeneration of such a constraint it can, for example, be used again to fire propagation rules, because it is then annotated with a different identifier.

After applying all the previous simplifications to the `twoloop` rule and transforming it into a simpagation rule we get the following simplified rule:

\[
\text{twoloop } @ \text{ node}(N_1, D_1) \setminus \text{ node}(N_2, 2), \text{ edge}(N_1, N_2), \text{ edge}(N_2, N_1) \iff \text{ edge}(N_1, N_1)
\]
The attentive reader might be tempted to always create simpagation rules in Definition 17.4, based on the idea that the context graph $K$ already identifies non-removed nodes. However, the above creation of simpagation rules with node constraints among the kept constraints, is only possible if the respective node's degree remains unchanged by the rule application.

**Propagation Rules** Readers more familiar with CHR may also wonder if propagation rules could be used as well. It is technically possible to define a GTS rule that does not remove any elements, but only adds new nodes and edges. However, a thusly created GTS would suffer from non-termination, i.e. such a rule could be applied infinitely often. As CHR implementations restrict propagation rule applications, however, they are not suitable to model the GTS's behavior. Instead we would have to use a simplification rule that introduces its rule head again in the body in order to remain faithful to the semantics of the GTS. When we use the equivalence-based operational semantics $\omega_e$, though, we may use propagation rules, because their implied trivial non-termination problem coincides with the intended semantics.

**Aggregates for Degrees** A completely different approach to determine a node’s degree for a rule application would be *aggregates*. Aggregates have been introduced in [Van Weert et al., 2008] and essentially allow us to count the number of adjacent edges of a node as part of the guard checking. For example, a rule in which a node with one adjacent edge should be deleted might then be of the following form.

$$n(N), e(E, N, X), \ldots \Leftrightarrow \text{edges}(N, _, C) \land C = 1 | \ldots$$

However, we refrain from using this aggregate-based approach for two reasons: Firstly, it is clearly more inefficient to continuously recalculate the degree of each node for which we investigate possible rule applications. Secondly and much more importantly, this assigns concrete degree values to each node, and hence, we can no longer consider partial graphs (cf. Section 18.2), which are the foundation for all our program analysis results.

**17.2.2 Example Derivation**

In this section, we provide a complete derivation for our cyclic list example to demonstrate how our encoding works. The following two rules are the CHR encoding of the rules in Figure II.3:

**unlink** $\forall$ node($N_1, D_1$), node($N_2, D_2$) \node($N, 2$), edge($E_1, N_1, N$), edge($E_2, N, N_2$)

$$\Leftrightarrow$$

edge($E, N_1, N_2$)

**twoloop** $\forall$ node($N_1, D_1$) \node($N, 2$), edge($E_1, N_1, N$), edge($E_2, N, N_1$)

$$\Leftrightarrow$$

edge($E, N_1, N_1$)

The following state encodes a cycle consisting of three nodes. The following derivation is depicted in Figure V.3. To demonstrate computations on so-called partial graphs (cf. Section 18.2), the degree of the third node is left uninstantiated.

$$\sigma = \langle \text{node}(N_1, 2), \text{node}(N_2, 2), \text{node}(N_3, D_3),$$

$$\text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_3), \text{edge}(E_3, N_3, N_1);$$

$$\top; \{N_1, N_2, N_3, E_1, E_2, E_3, D_3\} \rangle$$
Rule *unlink* is applied to state $[\sigma]$ resulting in the state

$$[\sigma] \rightarrow [(\text{node}(N_1, 2), \text{node}(N_3, D_3), \text{edge}(E, N_1, N_3), \text{edge}(E_3, N_3, N_1); \top; \{N_1, N_3, E_3, D_3\})].$$

Finally, rule *twoloop* is applied to the above state, which removes node $N_1$, resulting in

$$[(\text{node}(N_3, D_3), \text{edge}(E', N_3, N_3); \top; \{N_3, D_3\})].$$

Without the simplifications we made to the rules, the built-in store would contain a chain of degree adjustments for $D_3$ of the form $D_3'' = D_3' + 1 - 1, D_3' = D_3 + 1 - 1$. Such chains are always equivalent to a single built-in constraint of the form $D_3' = D_3 \pm k$, that relates the original degree $D_3$ to the resulting degree $D_3'$ of the current state. Other interesting consequences of partially uninstantiated encodings of host graphs are investigated more thoroughly in Section 18.2.

### 17.2.3 Platform for Analyzing GTSs

In the diploma thesis by Wasserthal [2009], an extensible platform for the analysis of graph transformation systems using Constraint Handling Rules is described. It is based on JCHR [Van Weert et al., 2005], a Java-based implementation of CHR, and the work presented in Section 17.

The developed tool presents a graphical view of a GTS, which is synchronized with the corresponding GTS-CHR program at all times. Furthermore, it provides an interface for program analysis plug-ins, which can work directly on the GTS or on the GTS-CHR program. The screenshot in Figure V.4 shows a type graph, for which a corresponding typed graph is being constructed. Additionally, the GUI offers a textual representation of all graphs and rules as CHR source code. Any changes made to either the visual or textual model are reflected to the other model.
18. Properties of Encoding

Implementation details and examples of program analyses, including a ranking-based termination analysis, are given in [Wasserthal, 2009].

18 Properties of Encoding

This section examines formal properties of the encoding given in Section 17. First, Section 18.1 analyzes the special CHR states found, when working with a GTS-CHR program. Next, we discuss partially defined graphs in Section 18.2, before we prove soundness and completeness of the encoding in Section 18.3.

18.1 Graph States

Our encoding is based on the assumption that the resulting CHR programs are executed only for initial states that correspond to graphs. We are not interested in executions for arbitrary CHR states. Hence, in this section we present a formal characterization of a CHR state that is the encoding of a graph. Furthermore, we compare the different equivalence notions of the two systems, i.e. graph isomorphism and CHR state equivalence.

In order to determine if a CHR state encodes a graph, we define a predicate that holds if and only if this is the case. It is intuitively clear, that starting with the encoding of a graph and transforming it via a graph transformation rule yields the encoding of a graph again. Although the below definition declares an invariant, we do not require it to be an invariant throughout this section, but solely consider it as a predicate. In Section 18.3, more precisely Corollary 18.7, we will then show that it is indeed a proper invariant.

Definition 18.1 (Graph Invariant). Let \( \sigma = \langle G; B_c \land B_a; V \rangle \) be a state where \( B_c \) are constraints of the form \( X = c \) for constants \( c \) and \( B_a \) are constraints of the form \( X = Y + c_1 - c_2 \) for constants \( c_1, c_2 \).

The graph invariant \( G \) holds for state \( \sigma \) if and only if there exists a graph \( G \) and a conjunction \( B \) of equality constraints of the form \( X = c \) for a variable \( X \) and constant \( c \), such that

\[
\langle G; B_c \land B_a \land B; \emptyset \rangle \equiv_e \langle \text{chr(ground}, G); \top; \emptyset \rangle
\]

For a state \( \sigma \), for which \( G(\sigma) \) holds with a graph \( G \), we say \( \sigma \) is a \( G \)-state based on \( G \).

We lift the definition of the invariant \( G \) to equivalence classes of states, assuming that it holds for a representant state of the above form.

Example 18.2. Consider again the following final state from the example derivation, given in Section 17.2.2.

\[
\langle \text{node}(N_3, D_3), \text{edge}(E', N_3, N_3); \top; \{N_3, D_3\} \rangle
\]

By using the equality constraint \( B := (D_3 = 2) \) the resulting state for Definition 18.1 is equivalent to

\[
\langle \text{node}(N_3, 2), \text{edge}(E', N_3, N_3); \top; \emptyset \rangle.
\]

Let \( G \) be the graph consisting of a node \( v \) with a loop, then

\[
\langle \text{chr(ground}, G); \top; \emptyset \rangle = \langle \text{node}(N_v, 2), \text{edge}(\tilde{E}, N_v, N_v); \top; \emptyset \rangle.
\]

Therefore, the invariant \( G \) is satisfied for the above state, as the corresponding states are equivalent by renaming of local variables.
This example further shows why the variable set \( V \) is disregarded for the two states. The variable given by \( \text{var} \) for a node of the graph has to coincide with the corresponding global variable for both states to be equivalent. Hence, for the above graph with node \( v \), knowledge of the original state would be required to determine that \( \text{var}(v) = N_3 \). Omitting global variables from both states, however, allows us to freely map \( v \) to any variable through \( \text{var}(v) \).

**Example 18.3.** For the state \( \sigma = \langle \text{chr}(\text{keep}, G); \top; V \rangle \) there clearly exists such a graph \( G \), for which \( B \) simply assigns the corresponding degree variables. States may also be in-between \( \text{chr}(\text{ground}, G) \) and \( \text{chr}(\text{keep}, G) \), in the sense that only a subset of the degree variables are instantiated, resulting in a state \( \sigma' = \langle \text{chr}(\text{keep}, G); B_c; V \rangle \) with \( B_c \) being the corresponding equality constraints. By instantiating the remaining degrees it is clear that \( G(\sigma') \) holds.

Arithmetic built-in constraints, introduced by bodies of rules in order to adjust a node’s degree, are covered by the above graph invariant definition: The introduction of the corresponding degree equality constraint leads to a collapse of the chain of arithmetic constraints. Hence, the concept of a \( G \)-state based on \( G \) also applies to intermediate computation states, which gives rise to the following lemma.

**Lemma 18.4 (Graph States).** Let \( G(\sigma) \) hold for a state \( \sigma \), then there exists a graph \( G \) such that

\[
\sigma \equiv_e \langle \text{chr}(\text{keep}, G); B_c \land B_a; V \rangle
\]

- \( B_c \) is a conjunction of \( \text{dvar}(v) = \text{deg}_G(v) \) constraints
- \( B_a \) is a conjunction of \( \text{dvar}(v') = \text{dvar}(v) + c_1 - c_2 \) constraints

*Proof.* Let \( \sigma = \langle G; B_c \land B_a; V \rangle \), then by Def. 18.1 we have that \( \langle G; B_c \land B_a \land B; \emptyset \rangle \equiv_e \langle \text{chr}(\text{ground}, G); \top; \emptyset \rangle \) for a graph \( G \) and \( X = k \) constraints \( B \).

W.l.o.g. all identifier variables occurring in \( \text{chr}(\text{ground}, G) \) (and therefore in \( \text{chr}(\text{keep}, G) \)) also occur in \( G \) as identifier variables. Due to the state equivalence the difference between \( G \) and \( \text{chr}(\text{keep}, G) \) can then only consist in \( G \) specifying some node degrees by constants (for degree variables we can again assume that they are the same as in \( \text{chr}(\text{keep}, G) \)).

Let \( \Theta \) be the conjunction of equality constraints of the form \( X = c \) for each degree specified explicitly in \( G \), using fresh variables for \( X \). Interpreting \( \Theta \) as a substitution, replacing \( X \) with \( c \) for each of the equivalences, we have that

\[
\sigma = \langle \text{chr}(\text{keep}, G)\Theta; B_c \land B_a; V \rangle.
\]

As all variables occurring in \( \Theta \) are local, we get by Def. 8.2:2 and 1 that

\[
\sigma \equiv_e \langle \text{chr}(\text{keep}, G)\Theta; B_c \land B_a \land \Theta; V \rangle \\
\equiv_e \langle \text{chr}(\text{keep}, G); B_c \land B_a \land \Theta; V \rangle \\
= \langle \text{chr}(\text{keep}, G); B'_c \land B_a; V \rangle.
\]

\( \square \)

The reverse direction of Lemma 18.4 does not hold in general: The state \( \sigma = \langle \emptyset; D = 0 \land X = 1 \land D = X + 2 - 0 \land \emptyset \rangle \) satisfies the conditions for an empty graph \( G \), but of course \( G(\sigma) \) does not hold, as \( \langle \emptyset; \bot; \emptyset \rangle \neq_e \langle \emptyset; \top; \emptyset \rangle \).

The following lemma presents an interesting fact about the correspondence between state equivalence and graph isomorphism: Equivalent CHR states encoding two graphs imply that
these graphs are isomorphic. This lemma was not contained in [Raiser and Frühwirth, 2009c],
but was facilitated by our proper state equivalence definition. With the equivalence-based
operational semantics as the foundation, we are now able to directly relate state equivalence
in CHR with graph isomorphism via the following lemma.

Lemma 18.5 (Equivalent $\mathcal{G}$-states imply Graph Isomorphism). Given a state $\sigma_1$ with
$\sigma_1 = \langle \text{chr(keep}, G_1); \mathbb{B}_1; \forall \rangle$, a $\mathcal{G}$-state based on $G_1$, and a state $\sigma_2 = \langle \text{chr(keep}, G_2); \mathbb{B}_2; \forall \rangle$, a $\mathcal{G}$-state based on $G_2$, then

$$\sigma_1 \equiv_e \sigma_2 \Rightarrow G_1 \cong G_2$$

Proof. First, we note that $\mathbb{B}_1, \mathbb{B}_2$ consist only of degree equalities or adjustments. Therefore,
we consider the following states instead, which are already sufficient to imply the isomorphism:

$$\langle \text{chr(keep}, G_1); \top; \forall \rangle \equiv_e \langle \text{chr(keep}, G_2); \top; \forall \rangle$$

W.l.o.g. let the local variables occurring in the two states be disjoint (it is clear that otherwise
we can consider equivalent states that only differ by renaming of local variables and that these
states all provide corresponding graph isomorphisms).

Let $\bar{y}_1$ and $\bar{y}_2$ be the set of local variables of the two states. We can then apply the criterion
from Thm. 3 to get

$$\mathcal{CT} \models \exists \bar{y}_1. \text{chr(keep}, G_1) = \text{chr(keep}, G_2)$$

As there are only variable terms contained in this equivalence we have the following conclusion,
where $c(\bar{t})$ is any constraint with argument terms, i.e. variables, $\bar{t}$.

$$\exists f : \bar{y}_1 \rightarrow \bar{y}_2 \text{ with } c(\bar{t}) \in \text{chr(keep}, G_1) \rightarrow c(f(\bar{t})) \in \text{chr(keep}, G_2)$$

We know that $f$ is surjective (as the variables are disjoint and the above equality demands
that at least one variable from $\bar{y}_1$ is mapped to each variable in $\bar{y}_2$). A consequence of this is that $|\bar{y}_1| \geq |\bar{y}_2|$.

Analogously, we get from $\mathcal{CT} \models \exists \bar{y}_2. \text{chr(keep}, G_1) = \text{chr(keep}, G_2)$ that $|\bar{y}_2| \geq |\bar{y}_1|$, and hence,

$|\bar{y}_1| = |\bar{y}_2|$. From this follows that $f$ is also injective, and therefore, bijective.

Next we realize, that by the above equality, $f$ has to map local variables corresponding to node
identifiers to local variables that also correspond to node identifiers. Let $\bar{y}_{n_1} \subset \bar{y}_1, \bar{y}_{n_2} \subset \bar{y}_2$ be
the local variables used for node identifiers, then $f' : \bar{y}_{n_1} \rightarrow \bar{y}_{n_2}, y \mapsto f(y)$ is a well-defined and
bijective function. We use this to define the actual graph isomorphism function $g : V_{G_1} \rightarrow V_{G_2}$
with $v \mapsto v'$ for $f'(\text{var}(v)) = \text{var}(v')$.

$g$ is well-defined: for every node there is a corresponding node identifier variable that $f'$ maps
to another local variable or the same global variable, as otherwise the $\equiv_e$ relation cannot
hold. Furthermore, $g$ is bijective as well, because it is defined bijectively via $f'$.

Finally, $g$ is a graph isomorphism: By the above equality we have corresponding pairs of
edge constraints. For every edge adjacent to a node identified by a local variable this variable
is bijectively mapped to another local variable and the above equality ensures that the
Corresponding edge is adjacent to the same node in order to satisfy $\equiv_e$. $\square$

The reverse direction of Lemma 18.5 cannot hold in general: The encoding of the graphs $G_1$
and $G_2$ are independent from determining the set $\forall$ of global variables. Even a graph consisting
of a single node only can be encoded in two ways, such that the states are not equivalent:

$$\langle \text{node}(\text{N}, 0); \top; \emptyset \rangle \not\equiv_e \langle \text{node}(\text{N}, 0); \top; \{\text{N}\} \rangle$$

108
18.2 Partial Graphs

In the example computation given in Section 17.2.2 the input contains a node with a variable degree: \( \text{node}(N_3, D_3) \). Nevertheless, computations on this input are possible and the example resulted in the final state

\[
\langle \text{node}(N_3, D_3), \text{edge}(E', N_3, N_3); \top; \{N_3, D_3\} \rangle.
\]

In general, a variable node degree will cause a chain of degree adjustment constraints to be created, i.e. constraints of the form \( X = Y + c_1 - c_2 \). These stem from the node being involved in a rule application that affects its degree.

We can only match such a node in rules that do not remove it. A rule that removes a node contains the explicit degree for it in the head, which cannot be matched by a variable degree. As a consequence, specifying variable degrees in the input ensures that the corresponding nodes will not be removed during the computation.

While this is an interesting feature in its own right, it provides the basis for many forms of program analysis. The aim of program analysis is to make statements for an infinite number of graphs, while only having to investigate a small selection of them. Graph encodings with variable degrees can here be thought of as partially defined graphs, i.e. there may be any number of further edges being connected to a node with a variable degree. Due to the importance of these nodes, we introduce the following definition.

**Definition 18.6 (Strong Nodes).** For a CHR state \( \sigma \equiv_\sigma \langle \text{chr(keep, } G); B; V \rangle \) which is a \( G \)-state based on \( G \), we define the set of strong nodes as:

\[
S(\sigma) = \{ v \in V_G \mid \text{dvar}(v) = \text{deg}_G(v) \not\in B \}
\]

Partially defined graphs only exist within the CHR context. In a GTS the degree of a node is implicitly given by the adjacent edges. As a consequence, having a strong node in the CHR encoding ensures, that it will not be removed during computation. In the GTS context, we have no such option available for host graphs.

By the above argument, the state

\[
\langle \text{node}(N, D); \top; \{N, D\} \rangle
\]

therefore not only represents the graph consisting of a single node and no edges. Instead, it represents the set of all graphs with at least one node. Similarly, the above final state from Section 17.2.2 stands for the set of graphs that contain at least one node with a loop.

Every derivation performed on an input with strong nodes actually represents derivations for an infinite set of graphs. This is a fundamental feature of our encoding for program analysis and will be exploited in Sections 19 and 20.

18.3 Soundness and Completeness

In this section, we prove soundness and completeness of our embedding. That \( G \) is an invariant for a GTS-CHR program and that termination of a GTS and its GTS-CHR program coincide, are then derived as consequences of the main theorem below.
Theorem 11 (Soundness and Completeness). Let $\sigma \equiv_e \langle \text{chr(keep, } G); B; V \rangle$ be a CHR state with $G(\sigma)$ holding with graph $G$. Then

$$G \Rightarrow^m H \text{ with } \{ v \in V_G \mid \text{tr}_{G \Rightarrow H}(v) \text{ defined} \} \supseteq S(\sigma)$$

if and only if

$$\sigma \not\Rightarrow^e_r \tau = \langle \text{chr(keep, } H); B'; V' \rangle \text{ and } G(\tau) \text{ holds with graph } H.$$  

Proof. In order to shorten this proof we use $k(G)$ and $g(G)$ to denote chr(keep, $G$) and chr(ground, $G$), respectively.

$a \Rightarrow^m \Rightarrow$:

Let $G \Rightarrow^m_H$ and let $r : L \leftarrow K \to R$.

Let $G := k(G) = k(G \setminus m(L)) \cup k(m(E_L)) \cup k(m(V_K)) \cup k(m(V_L \setminus V_K)) \Rightarrow \sigma \equiv_e \langle G; B; V \rangle$.

Let $g(r) = (r \oplus C_L \iff C_R^b, C_R^b)$ with $C_L = k(K) \cup g(L \setminus K)$.

Let $v \in V_L$, then $\text{type}_{G}(v)(\text{var}(v), \ldots) \in C_L \land \text{type}_{G}(v)(\text{var}(m(v)), \text{dvar}(m(v))) \in k(m(V_K))$,

as the types match due to $m$ being a graph morphism.

As we have a fresh rule using node $v$ that does not occur elsewhere, we can say by Def. 8.2:2 that $\sigma \equiv_e \langle G; \text{var}(m(v)) = \text{var}(v) \land B; V \rangle$, and hence, by Def. 8.2:1

$$\sigma \equiv_e \langle G[\text{var}(m(v)) / \text{var}(v)]; \text{var}(m(v)) = \text{var}(v) \land B; V \rangle. \quad (V.1)$$

Consider $v \in V_L \setminus V_K$, then $\text{type}_{G}(v)(\text{var}(v), \text{deg}_{L}(v)) \in C_L$. Assume that $m(v) \in S(\sigma)$, then $\text{tr}_{G \Rightarrow H}(m(v))$ is defined, which is a contradiction to $v \in V_L \setminus V_K$. Therefore, $m(v) \notin S(\sigma)$ and hence $\text{dvar}(m(v)) = \text{deg}_{G}(m(v)) \in B$. As $G \Rightarrow^m H$ satisfies the gluing condition, we know that $\text{deg}_{L}(v) = \text{deg}_{G}(m(v))$.

Therefore, we have by Def. 8.2:1 that

$$\sigma \equiv_e \langle G[\text{var}(m(v))/\text{var}(v)]; \text{var}(m(v)) = \text{var}(v) \land B; V \rangle.$$  

From (V.1) for nodes $v \in V_K$ and the above for nodes $v \in V_L \setminus V_K$ follows for a conjunction of equality constraints $E$ that

$$\sigma \equiv_e \langle k(G \setminus m(L)), k(m(E_L)), k(V_K), g(V_L \setminus V_K); B \land E; V \rangle = \langle G'; B \land E; V \rangle.$$  

Let $e \in E_L$, then type$_{G}(e)(\text{var}(e), \text{var}(\text{src}(e)), \text{var}(\text{tgt}(e))) \in C_L$ and after the previous substitutions have been made for node identifier variables, and as $k(e) = g(e)$, we get type$_{G}(m(e))(\text{var}(m(e)), \text{var}(\text{src}(e)), \text{var}(\text{tgt}(e))) \in \sigma$. We then have

$$\sigma \equiv_e \langle G'[\text{var}(m(e))/\text{var}(e)]; \text{var}(m(e)) = \text{var}(e) \land B \land E'; V \rangle. \quad (V.2)$$

By applying this substitution for all edges $e \in E_L$ and extending $E$ with the required equalities to $E'$ we get

$$\sigma \equiv_e \langle k(G \setminus m(L)), k(E_K), g(E_L \setminus E_K), k(V_K), g(V_L \setminus V_K); B \land E'; V \rangle.$$  

Hence, $\sigma \equiv_e \langle k(G \setminus m(L)), C_L; B \land E' \rangle$ such that we apply the rule $g(r)$ to $\sigma$:

$$[\sigma] \not\Rightarrow^e_r \tau = \langle \langle k(G \setminus m(L)), C_R^b; B \land E' \land C_R^b; V \rangle \rangle = \langle \langle k(G \setminus m(L)), g(R \setminus K), k(E_K), k(V_K \setminus K); B \land E' \land C_R^b; V \rangle \rangle.$$
As $C^b_R$ contains $\text{var}(v') = \text{var}(v) \forall v \in V_K$ let $C'_R$ be $C^b_R$ without these constraints, then
\[
\tau \equiv_e (k(G \setminus m(L)), g(R \setminus K), k(E_K), k(V_K); \mathbb{B} \land E' \land C'^b_R; \forall) \\
\equiv_e (k(G \setminus m(L)), g(R \setminus K), k(E_K), k(V_K); \mathbb{B} \land E' \land C'_R; \forall).
\]

Let $\hat{G} := k(G \setminus m(L)) \cup k(K)$, then $\tau \equiv_e (\hat{G}, g(R \setminus K); \mathbb{B} \land E' \land C'_R \land D_R; \forall)$ with $\forall v \in V_R \setminus V_K$, $\text{dvar}(v) = \text{deg}_R(v) \in D_R$. Furthermore, consider $\Theta$ a substitution corresponding to the reversed reading of $E'$ which undoes the ideas of (V.1) and (V.2) for all affected nodes and edges. We then get
\[
\tau \equiv_e (\hat{G}, k(R \setminus K); D_R \land C'_R \land \mathbb{B} \land E'; \forall) \\
\equiv_e (k(G \setminus m(L \setminus K)), (k(R \setminus K)\Theta); D_R \land C'_R \land \mathbb{B} \land E'; \forall) \\
\equiv_e (k(G \setminus m(L \setminus K)), (k(R \setminus K)\Theta); D_R \land C'_R \land \mathbb{B}; \forall) \\
\equiv_e (k(\hat{H}); \mathbb{B}; \forall).
\]

We get the graph $H$ as its DPO construction corresponds to the removal of $m(L \setminus K)$ and addition of $R \setminus K$. $\Theta$ is needed to attach the new nodes of $R \setminus K$ to nodes from $V_K$ and $C'_R$ contains degree adjustments for those nodes that are correct by construction. Hence, it also holds that $\mathcal{G}(\tau)$ is satisfied with graph $H$.

\[\text{“} \Leftarrow \text{”}\]:

Let $[\sigma] \rightarrow^e [\tau]$ with $\tau \equiv_e (k(\hat{H}); \mathbb{B}; \forall)$ and $\mathcal{G}(\tau)$ holds with graph $H$. Let $g(r) = (r \otimes C_L \Leftrightarrow C^b_R, C'_R)$ with $C_L = k(K) \cup g(L \setminus K)$. From Def. 8.6 follows that
\[
\sigma \equiv_e (k(K), g(L \setminus K), k(G \setminus L); \mathbb{B}_1; \forall).
\]

Using Lemma 18.4 and with $E$ being a conjunction of $\text{var}(m(x)) = \text{var}(x)$ constraints for $x \in L$, we get
\[
\sigma \equiv_e (k(G); \mathbb{B}_c \lor \mathbb{B}_a; \forall) \\
\equiv_e (k(K), k(L \setminus K), k(G \setminus m(L)); \mathbb{B}_c \lor \mathbb{B}_a \land E; \forall) \\
\equiv_e (k(K), g(L \setminus K), k(G \setminus m(L)); \mathbb{B}_c \lor \mathbb{B}_a \land E'; \forall),
\]

where $E'$ is the extension of $E$ with $\text{dvar}(m(v)) = \text{deg}_G(v)$ constraints for $v \in V_L \setminus V_K$ and $\mathbb{B}_1 = \mathbb{B}_c \lor \mathbb{B}_a \land E'$. $m : L \rightarrow G$ is well-defined and injective by the multiset semantics of CHR and it remains to be shown, that $m$ is a graph morphism. Therefore, let $e \in E_L$, then

\[
\text{type}_L(e)(\text{var}(e), \text{var}(\text{src}(e)), \text{var}(\text{tgt}(e))) \in C_L \land \\
\text{type}_L(\text{src}(e))(\text{var}(\text{src}(e)), \_), \text{type}_L(\text{tgt}(e))(\text{var}(\text{tgt}(e)), \_) \in C_L.
\]

It follows that $\text{var}(m(e)) = \text{var}(e)$, $\text{var}(m(\text{src}(e))) = \text{var}(\text{src}(e))$, and $\text{var}(m(\text{tgt}(e))) = \text{var}(\text{tgt}(e))$ are all in $\mathbb{B}_1$. Therefore, $m(\text{src}(e)) = \text{src}(m(e)) \land m(\text{tgt}(e)) = \text{tgt}(m(e))$.

The gluing condition is satisfied, as $\forall v \in V_L \setminus V_K$ the matched degree ensures that there are no dangling edges, hence, $r$ is GTS-applicable to $G$. Similarly to the other proof direction, we show that the DPO construction of $H$ coincides with the construction of $\tau$ by the CHR rule application:

111
\[ \sigma \mapsto^\tau [r] = \langle k(K), g(R \setminus K), k(G \setminus m(L)), B_c, B_a \rangle \setminus E' \land C'_{R'; \emptyset} \vDash \]

\[ = \langle k(m(K)), g(R \setminus K), k(G \setminus m(L)), B_c, B_a \rangle \setminus E \land C_0^{R'; \emptyset} \vDash \]

\[ = \langle g(R \setminus K), k(G \setminus m(L \setminus K)), B_c, B_a \rangle \setminus E \land C_0^{R'; \emptyset} \vDash \]

\[ = \langle k(H), B_c \setminus B_a \rangle \setminus E \land C_0^{R'; \emptyset} \vDash \]

where \( \Theta \) is the reversed substitution for \( E \) similar to the other proof direction. The final equivalence comes from extracting the degrees of constraints in \( g(R \setminus K) \) into equality constraints contained in \( B' \). As can be seen here, the application of the rule results in a state encoding the graph \( H \), such that \( G_{r,m} \Rightarrow H \) holds.

Finally, for the set \( S(\sigma) \) we know that the nodes cannot be removed by rule \( r \): For a node \( v \in V_c \setminus V_K \) we have \( \text{type}_L(v)(\text{var}(v), \text{deg}_L(v)) \in C_L \), but this cannot be matched with \( \sigma \), as by Def. 18.6 the corresponding degree is unavailable. Hence, none of the nodes from \( S(\sigma) \) are removed by the rule application \( G_{r,m} \Rightarrow H \), i.e. \( \text{tr}_{G_{r,m}}(H)(v) \) is defined for all \( v \in S(\sigma) \).

Excursion: Comparison to [Raiser, 2007]

The original proof given in [Raiser, 2007] differs from the above proof in the following aspects.

**Generality** The above proof explicitly takes partial graphs into account, whereas the original proof assumed all degrees to be known. In Theorem 11, we further work with arbitrary graph states, whereas the original theorem assumes the built-in store to be the empty conjunction.

**Length** Our proof above did not require any additional lemmata beyond what we have established for CHR in general in Chapter III and our insight into graph states given in Lemma 18.4. Contrary to that, in [Raiser, 2007] significant effort was required to prove applicability of rules. For a direct comparison of the proof lengths, consider that in [Raiser, 2007] the second direction is not explicitly given and the proof was written in a more dense style due to size restrictions. Taking this into account, we roughly reduced proof length to one third for Theorem 11.

**Conciseness** In [Raiser, 2007], the proof contained several textual explanations, mostly because at that time we lacked the formal tools to express these statement in a concise fashion. In contrast, the proof of Theorem 11 reduces textual elements mostly to support the reader by explaining the proof structure.

As the proof of Theorem 11 reveals, a GTS-CHR rule application on a \( G \)-state based on \( G \) always results in a state encoding a corresponding graph \( H \), which gives us the following corollary.

**Corollary 18.7** (Invariant \( G \)). For a GTS-CHR program \( G \) is an invariant.

**Proof.** Follows directly from Theorem 11. \( \square \)
A closer look at the conditions required in Theorem 11 reveals that for a state \( \sigma \) with \( S(\sigma) = \emptyset \), i.e. for an encoding of a graph with all degrees explicitly given, we have unrestricted soundness and completeness.

**Corollary 18.8** (Unrestricted Soundness and Completeness). Let \( \sigma \) be a CHR state with \( \sigma \equiv_e (\text{chr(ground}, G); \top; V) \) and \( G(\sigma) \) holding with graph \( G \). Then

\[
G \xrightarrow{r,m} H
\]

if and only if

\[
[\sigma] \xrightarrow{e} [\tau] = [(\text{chr(ground}, H); \top; V)] \text{ and } G(\tau) \text{ holds with graph } H.
\]

**Proof.** This follows from Theorem 11 and the following insight: as all degrees of \( G \) are specified explicitly and all nodes added by the rule are also given explicit degrees, all degrees in \( H \) are given explicitly as well, which allows us to use \( \text{chr(ground}, H) \) here.

Finally, the soundness and completeness result induces a termination correspondence between a GTS and its GTS-CHR program. Again, we restrict our observation to graph-encoding states.

**Corollary 18.9** (Termination Correspondence). A GTS is terminating if and only if its corresponding GTS-CHR program is \( G \)-terminating, i.e. terminating for all \( G \)-states.

**Proof.** If a GTS contains a non-terminating derivation, we have the corresponding computation in its GTS-CHR program by Corollary 18.8. Similarly, if the GTS-CHR program has a non-terminating computation, there exists a corresponding non-terminating GTS derivation according to Theorem 11.

19 Confluence Analysis

After investigating our encoding of GTSs in CHR, we now compare program analysis in both systems. As mentioned earlier, a particularly interesting property is confluence, which has been introduced in Section 7.2. It is a worthwhile subject, because we can decide confluence of a terminating CHR program and one might expect that another formalism with a sound and complete embedding in CHR should display the same behavior. However, Plump [2005] proved that for a terminating graph transformation system, the confluence property remains undecidable.

Hence, an interesting question arises: What happens, if we take a terminating GTS, encode it in CHR, and then determine confluence of this GTS-CHR program? As termination is preserved by our encoding, confluence of the GTS-CHR program is decidable and in this section we will analyze what the results of the CHR confluence check imply for the original GTS.

First, Section 19.1 examines the relation between critical pairs of a GTS and its corresponding GTS-CHR program. Next, in Section 19.2 we apply our results on invariant-based confluence from Section 14 in order to decide confluence of a GTS through analyzing confluence of its corresponding GTS-CHR program.

Although confluence of a terminating GTS is undecidable, we show the decidable invariant-based confluence test of the GTS-CHR program coincides with strong joinability analysis of
Lemma 19.1
result (cf. Theorem 11), we later transfer joinability results to the critical GTS pair.

Correspondence of Critical Pairs

The discrepancy in decidability of the two systems’ confluence properties is discussed in Section 19.3 for exemplary critical pair analyses.

19.1 Correspondence of Critical Pairs

We now investigate the difference between critical GTS pairs and critical CHR pairs for GTS-CHR programs. The following lemma shows that there exists a corresponding overlap for each critical GTS pair. Therefore, by examining the overlaps and applying the previous soundness result (cf. Theorem 11), we later transfer joinability results to the critical GTS pair.

Lemma 19.1 (Overlap for Critical GTS Pair). If $P_1 \triangleright_{\text{term}} G \triangleright_{\text{term}} P_2$ is a critical GTS pair, then there exists an overlap $\sigma_C$ of $\varphi(v_1) = (r_1 @ C_{L1} \Rightarrow C_{L1}^u, C_{R1}^b)$ and $\varphi(v_2) = (r_2 @ C_{L2} \Rightarrow C_{R2}^u, C_{R2}^b)$ which is a $\mathcal{G}$-state based on $G$ and a critical CHR pair $(\sigma_1, \sigma_2)$ such that $\sigma_1$ is a $\mathcal{G}$-state based on $P_1$ and $\sigma_2$ is a $\mathcal{G}$-state based on $P_2$.

Proof. Let the two GTS rules be $L_i \leftarrow K_i \rightarrow R_i$ for $i = 1, 2$ and let $M = m_1(L_1) \cap m_2(L_2)$. We then define the following sets of constraints from which we construct the overlap:

$$
H_1 = \{ \text{chr}_{L_1}(\text{keep}, x) \mid x \in L_1 \land m_1(x) \notin M \}
$$

$$
H_2 = \{ \text{chr}_{L_2}(\text{keep}, x) \mid x \in L_2 \land m_2(x) \notin M \}
$$

$$
A_1 = \{ \text{chr}_{L_1}(\text{keep}, x) \mid x \in L_1 \land m_1(x) \in M \}
$$

$$
A_2 = \{ \text{chr}_{L_2}(\text{keep}, x) \mid x \in L_2 \land m_2(x) \in M \}
$$

$$
C_1 = \{ \text{dvar}(v) = \text{deg}_{L_1}(v) \mid v \in V_{L_1} \setminus V_{K_1} \}
$$

$$
C_2 = \{ \text{dvar}(v) = \text{deg}_{L_2}(v) \mid v \in V_{L_2} \setminus V_{K_2} \}
$$

Let $V = \text{vars}(H_1 \cup H_2 \cup A_1 \cup A_2)$ and let $\sigma = \langle H_1; C_1; V \rangle$, then $\sigma \equiv_e \sigma'$ with $\sigma' = \langle \{ \text{chr}_{L_1}(\text{keep}, x) \mid x \in K_1 \land m_1(x) \notin M \} \cup \{ \text{chr}_{L_1}(\text{ground}, x) \mid x \in L_1 \setminus K_1 \land m_1(x) \notin M \}; V; V \rangle$. By Def. 17.4 we have that $H'_1 \cup A'_1 = C_{L_1}$ and $H'_2 \cup A'_2 = C_{L_2}$. As $M \neq \emptyset$ it follows that $A'_1$ and $A'_2$ are non-empty. To investigate, if $CT \models \exists(A'_1 = A'_2)$ we take a closer look at the equality constraints imposed by $A'_1 = A'_2$:

- $\{ \text{var}(v_1) = \text{var}(v_2) \mid v_1 \in V_{L_1} \land v_2 \in V_{L_2}, m_1(v_1) = m_2(v_2) \}$
- $\{ \text{dvar}(v_1) = \text{dvar}(v_2) \mid v_1 \in V_{K_1} \land v_2 \in V_{K_2} \land m_1(v_1) = m_2(v_2) \}$
- $\{ \text{dvar}(v_1) = \text{deg}_{L_1}(v_2) \mid v_1 \in V_{K_1} \land v_2 \in V_{L_2} \land V_{K_2} \land m_1(v_1) = m_2(v_2) \}$
- $\{ \text{var}(e_1) = \text{var}(e_2) \mid e_1 \in E_{K_1} \land e_2 \in E_{K_2} \land m_1(e_1) = m_2(e_2) \}$
- $\{ \text{deg}_{L_1}(v_1) = \text{deg}_{L_2}(v_2) \mid v_1 \in V_{L_1} \land V_{K_1} \land v_2 \in V_{L_1} \land V_{K_2} \land m_1(v_1) = m_2(v_2) \}$

Except for the last row, the above equality constraints can easily be satisfied under existential quantification. Hence, the only remaining problematic case is when two node constraints with constant degrees are overlapped. However, the degree of $m_1(v_1) = m_2(v_2)$ equals the degree
Figure V.5: Graph production rule for removing a loop

of \( v_1 \) and the degree of \( v_2 \) due to the gluing condition being satisfied, such that this case can only occur with equal constant degrees.

Hence, \( \sigma_{CP} = \langle H'_1 \sqcup A'_1 \sqcup H'_2; A'_1 = A'_2; V \rangle \) is an overlap of \( \varrho(r_1) \) and \( \varrho(r_2) \) with the critical CHR pair \( ((C^b_{R1} \sqcup H'_2; A'_1 = A'_2 \land C^b_{R1}; V); \langle C_{R2} \sqcup H'_1; A'_1 = A'_2 \land C^b_{R2}; V \rangle) \).

If we try to directly transfer the confluence property of a GTS to the corresponding GTS-CHR program, we cannot succeed however, as in general there are too many critical CHR pairs that could cause the GTS-CHR program to become non-confluent. The following example provides a rule which only has one critical GTS pair, but for which the corresponding CHR rule has three critical CHR pairs.

**Example 19.2.** Consider the graph production rule in Figure V.5. It removes a loop from a node and has the following corresponding CHR rule:

\[
R @ node(N, D), edge(E, N, N) \leftrightarrow node(N, D'), D' = D - 2
\]

To investigate confluence one must overlap this rule with itself which yields the following three CHR overlap states:

1. \( \langle node(N, D), edge(E, N, N), edge(E', N', N'); N = N'; \{ N, D, E, E', N' \} \rangle \)
2. \( \langle node(N, D), node(N', D'), edge(E, N, N); N = N'; \{ N, D, N', D', E \} \rangle \)
3. \( \langle node(N, D), edge(E, N, N); \top; \{ N, D, E \} \rangle \)

State (1) is not critical, because the corresponding graph transformations are parallel independent (cf. [Ehrig et al., 2006]), and hence, directly joinable by applying the rule again. State (2) is an invalid state, i.e. it violates \( \mathcal{G} \), as it has multiple encodings of the same node and state (3) is the encoding of the corresponding critical pair for the graph production rule.

### 19.2 Deciding Confluence via Embedding

In this section, we investigate how to decide confluence of a GTS via invariant-based confluence of its corresponding GTS-CHR program.

First, let us recall the results on invariant-based confluence from Section 14: For a terminating program, we can decide invariant-based confluence through investigation of critical pairs and all its invariant-satisfying extensions. A requirement for this, however, is that \( \ll \) is well-founded. Although, in our programs built-in constraints \( + \) and \( - \) occur, we can consider \( \ll \) well-founded for the following reason: On state components, other than the built-in store, the \( \ll \)-relation corresponds to the well-founded subset ordering with the minimal element \( \emptyset \) (cf. [Duck et al., 2007]). For the built-ins, we can consider \( + \) and \( - \) as successor/predecessor terms (as they are only used with constants in rules), and hence, we get well-foundedness via proposition 1 of [Duck et al., 2007].
A remaining problem in applying our previous results is the potentially infinite size of the set of minimal elements. However, the following lemma shows that in the context of GTS-CHR programs this situation cannot occur. We already know that \([\sigma_0]\) is the minimal element of the partial order \(<\). Therefore, for a state \(\sigma\) with \(G(\sigma)\) holding, we know that the only minimal element is \([\sigma_0]\). The lemma below considers the case that \(G(\sigma)\) is violated and shows that this invalidation cannot be fixed with minimal elements, hence, the set of minimal elements that could help satisfy the invariant is the empty set.

**Lemma 19.3 (No Minimal Elements).** If \(G(\sigma_{CP})\) is violated for an overlap \(\sigma_{CP}\), then no state \(\sigma_e\) exists such that \(G(\sigma_{CP} \circ \sigma_e)\) is satisfied, i.e. \(\Sigma^G([\sigma_{CP}]) = M^G([\sigma_{CP}]) = \emptyset\).

**Proof.** We proof this by a structural analysis of the overlap which gives the different possibilities for \(G(\sigma_{CP})\) to be violated. W.l.o.g. the overlap stems from the two rules \(\varrho(r_1) = (r_1 @ C_{L_1} \iff C_{u_1}^{w_1}, C_{d_1}^{b_1})\) and \(\varrho(r_2) = (r_2 @ C_{L_2} \iff C_{u_2}^{w_2}, C_{d_2}^{b_2})\) with the corresponding rule graphs \(L_1, L_2, K_1, K_2, R_1,\) and \(R_2\).

First consider the case of nodes \(v_1\) and \(v_2\) being overlapped:
Let \(\text{type}_{L_1}(v_1)(\var{v_1}, D_1) \in C_{L_1}\) and \(\text{type}_{L_2}(v_2)(\var{v_2}, D_2) \in C_{L_2}\) be overlapped with \(\text{type}_{L_1}(v_1) = \text{type}_{L_2}(v_2)\). The equality constraint \(\var{v_1} = \var{v_2} \in \sigma_{CP}\) resembles the merging of the two graph nodes \(v_1\) and \(v_2\). However, for the degree equalities different possibilities exist:

- \(D_1\) and \(D_2\) are constants: Then \(D_1 = D_2 = \deg_{L_1}(v_1) = \deg_{L_2}(v_2) = k\), as the overlap is impossible otherwise. Then \(\sigma_{CP}\) contains only one constraint

  \[
  \text{type}_{L_1}(v_1)(\var{v_1}, \deg_{L_1}(v_1)).
  \]

  As in \(L_1\) and \(L_2\) the nodes each have \(k\) adjacent edges, all constraints corresponding to adjacent edges in both rule graphs have to be contained in the overlap as well. If at least one such constraint is not part of the overlap, then \(\sigma_{CP}\) contains more than \(k\) constraints corresponding to edges adjacent to \(v_1 = v_2\). As the degree for the node is a constant, it cannot be changed by any extension and the additional edge constraints cannot be removed either. Therefore in such a case, no extension \(\sigma_e\) can correct the degree inconsistency and \(G(\sigma_{CP} \circ \sigma_e)\) cannot hold.

- \(D_1\) and \(D_2\) are variable: In this case the overlap is possible without any problems. Depending on the number of overlapped adjacent edge constraints the degree variables can always be instantiated with the correct degree, thus satisfying the invariant \(G\).

- w.l.o.g. \(D_1 = k\) and \(D_2\) is a variable: this means \(D_2 = k \in \sigma_{CP}\), therefore, all edge constraints of \(C_{L_2}\) of edges adjacent to \(v_2\) have to be overlapped with edge constraints of \(C_{L_1}\) corresponding to edges adjacent to \(v_1\). If there is such an edge constraint from \(C_{L_2}\) which is not contained in the overlap, then \(\sigma_{CP}\) contains more than \(k\) edge constraints corresponding to edges adjacent to \(v_1\). Again the degree of \(v_1\) is specified as the constant \(k\) in \(\sigma_{CP}\), and thus, an extension cannot correct this degree inconsistency.

If however, all these edge constraints are contained in the overlap, \(G\) is satisfied again, as there are exactly \(k\) such edge constraints coming from \(C_{L_1}\).

Finally, consider an edge being overlapped: Let

\[
\text{type}_{L_1}(\var{e_1}, \var{src(e_1)}, \var{tgt(e_1)}) \in C_{L_1}\) and
\text{type}_{L_2}(\var{e_2}, \var{src(e_2)}, \var{tgt(e_2)}) \in C_{L_2},
\]
then \( \var(e_1) = \var(e_2) \land \var(\text{src}(e_1)) = \var(\text{src}(e_2)) \land \var(\text{tgt}(e_1)) = \var(\text{tgt}(e_2)) \in \sigma_{\text{CP}} \). By Def. 17.4 we have constraints
\[
\text{type}_{L_1}(\text{src}(e_1))(\var(\text{src}(e_1)), \_)) \in C_{L_1} \quad \text{and} \quad \\
\text{type}_{L_2}(\text{src}(e_2))(\var(\text{src}(e_2)), \_)) \in C_{L_2}.
\]
If these two constraints are not part of the overlap, the corresponding equality constraint \( \var(\text{src}(e_1)) = \var(\text{src}(e_2)) \in \sigma_{\text{CP}} \) results in a single graph node being represented by two constraints. This is a violation of \( G \), as chr(ground, \( G \)) contains exactly one constraint for each node. This violation cannot be fixed by an extension, as the conflicting additional node constraint cannot be removed. Analogously, the two node constraints corresponding to \( \text{tgt}(e_1) \) and \( \text{tgt}(e_2) \) have to be contained in the overlap.

Therefore, an overlap \( \sigma_{\text{CP}} \) which violates the invariant \( G \) has to violate it due to one of the above reasons for which it cannot be extended by an extension \( \sigma_e \) such that \( G(\sigma_{\text{CP}} \circ \sigma_e) \) is satisfied.

Combining these two results, yields the criterion in Corollary 19.4 for deciding \( G \)-local-confluence. It is essentially the same criterion as used for traditional local confluence, except that the invariant \( G \) restricts the set of investigated overlaps.

**Corollary 19.4** (Deciding \( G \)-Local-Confluence). \( \mathcal{P} \) is \( G \)-local-confluent if and only if for all critical pairs \((\sigma_1, \sigma_2)\) with overlap \( \sigma_{\text{CP}} \), for which \( G(\sigma_{\text{CP}}) \) holds, we have \(((\sigma_1) \downarrow (\sigma_2))\).

**Proof.** This follows from the combination of Lemma 14.11, Lemma 19.3 and the insight that \([\sigma_0]\) is the unique minimal element of \(<\) in case of \( G(\sigma_{\text{CP}}) \) holding.

Next we transfer the joinability of critical CHR pairs to strong joinability in GTS.

**Lemma 19.5** (\( G \)-Confluence Implies Strong Joinability). If a \( G \)-terminating GTS-CHR program is \( G \)-confluent, then all critical GTS pairs are strongly joinable.

**Proof.** Let \( P_1 \overset{r_1 \leftarrow m_1}{\Rightarrow} G \overset{r_2 \rightarrow m_2}{\Rightarrow} P_2 \) be a critical GTS pair. Let \( r_i = (L_i \leftarrow K_i \rightarrow R_i) \) and \( g(r_i) = (r_i @ C_{L_i} \Rightarrow C'^R_{R_i}, C'^b_{R_i}) \) for \( i = 1, 2 \).

By Lemma 19.1 there exists an overlap \( \sigma_{\text{CP}} \) which is a \( G \)-state based on \( G \). As the critical pair \((\sigma_1, \sigma_2)\) created by the overlap \( \sigma_{\text{CP}} \) is joinable we have the computations \([\sigma_{\text{CP}}] \rightarrow_e [\sigma_1] \rightarrow^*_e [\tau] \) and \([\sigma_{\text{CP}}] \leftarrow_e [\sigma_2] \rightarrow^*_e [\tau] \). From Thm. 11 we know that there exist corresponding GTS transformations \( G \overset{r_1 \leftarrow m_1}{\Rightarrow} P_1 \Rightarrow^* X_1 \simeq X_2 \overset{r_2 \rightarrow m_2}{\Rightarrow} G \). The isomorphism between \( X_1 \) and \( X_2 \) follows from Lemma 18.5. Hence, the critical GTS pair is joinable.

To see that it is strongly joinable consider the set \( S(\sigma_{\text{CP}}) \). Every node \( v \), for which tr\(_G \Rightarrow P_1(v)\) and tr\(_G \Rightarrow P_2(v)\) are defined, is a node which is not deleted by either \( r_1 \) or \( r_2 \). As \( m_1 \) and \( m_2 \) are jointly surjective w.l.o.g. there exists a node \( v' \in V_{L_1} \) of rule \( r_1 \) with \( m(v') = v \). As the node is not removed we know \( v' \in V_{K_1} \), and therefore, \( \text{type}_{K_1}(v')(\var(v'), \text{dvar}(v')) \in C_{L_1} \).

Either the node is not part of the overlap in \( \sigma_{\text{CP}} \), or if it is overlapped with a node \( v'' \in V_{L_2} \) such that \( m(v') = m(v'') \), then we also know that \( v'' \in V_{K_2} \) due to the defined track morphism. Therefore, we always have the node constraint \( \text{type}_{K_1}(v')(\var(v'), \text{dvar}(v')) \in \sigma_{\text{CP}} \) and \( v \in S(\sigma_{\text{CP}}) \). As this node cannot be removed during the transformation, a variant of this constraint with adjusted degree is also present in \( \tau \). These two variant constraints are uniquely determined, as \( \var(v) \in \mathbb{V} \) by Def. 14.5, and hence, they both have to use \( \var(v) \) for the node identifier variable. This means, we still have to show for such a node \( v \) that the two conditions from Def. 7.2 are satisfied:
1. \( \text{tr}_G \Rightarrow P_1 \Rightarrow X_1(v) \) and \( \text{tr}_G \Rightarrow P_2 \Rightarrow X_2(v) \) are defined:

By Thm. 11 we know that the GTS transformations are strong w.r.t. \( S(\sigma_{CP}) \). As \( v \in S(\sigma_{CP}) \) this implies \( v \in m(K) \lor v \notin m(L) \) for each of the applied rules, i.e. the node remains during the transformation, and hence, the track morphisms are defined as in Def. 6.7.

2. \( f_V(\text{tr}_G \Rightarrow P_1 \Rightarrow X_1(v)) = \text{tr}_G \Rightarrow P_2 \Rightarrow X_2(v) \):

As the isomorphism \( f \) is derived from Lemma 18.5 and \( \text{var}(v) \in V \), this isomorphism correctly relates the original node \( v \) with its occurrences in \( \tau \).

The reverse direction holds as well, given a restriction on the joinability of the critical GTS pairs.

**Lemma 19.6 (Strong Joinability Implies \( G \)-Confluence).** If all critical GTS pairs of a terminating GTS are strongly joinable such that for each node the track morphism is either defined for both derivations, or neither, then the corresponding GTS-CHR program is \( G \)-confluent.

**Proof.** Consider an overlap \( \sigma_{CP} \) for the critical CHR pair \((\sigma_1, \sigma_2)\). W.l.o.g. \( G(\sigma_{CP}) \) holds according to Cor. 19.4. Therefore, \( \sigma_{CP} \) is a \( G \)-state based on \( G \) and \( \sigma_1, \sigma_2 \) correspond to graphs \( G_1, G_2 \). Consider now \( G_1 \xleftarrow{r_1} G \xrightarrow{r_2} G_2 \).

We now show, that either the critical CHR pair is non-critically joinable, or it corresponds to a critical GTS pair and can thus be joined, because all critical GTS pairs are strongly joinable.

First, we want to point out that \( G \) is minimal by the definition of the CHR overlap, i.e. every occurring node and edge is part of a match, hence, \( m_1 \) and \( m_2 \) are jointly surjective.

Next, we distinguish two cases: First, let \( G_1 \xleftarrow{r_1} G \xrightarrow{r_2} G_2 \) be parallel independent. Therefore, the second rule can be applied after the first, because none of the required nodes or edges has been removed. The following diagram depicts this situation:

```
  G
 /\   \
G_1  G_2
  \/<\  <
   X
```

By Theorem 11, we can apply the corresponding rules to \((\sigma_1, \sigma_2)\) in order to join the critical CHR pair, because \( S(\sigma_{CP}) \) contains only nodes not deleted by \( r_1 \) and \( r_2 \).

Secondly, let \( G_1 \xleftarrow{r_1} G \xrightarrow{r_2} G_2 \) be parallel dependent. It follows that \( m(L_1) \cap m(L_2) \not\subseteq m(K_1) \cap m(K_2) \). However, this is now a critical GTS pair, and hence, strongly joinable as depicted on the left of the following diagram:

```
  G
 /\   \
G_1  G_2
  \/<\  <
   X_1 \simeq X_2
```

\( \sigma_{CP} \)

```
  G
 /\   \
G_1  G_2
  \/<\  <
   \sigma_1 \equiv \sigma_2
```

**\( \square \)**
The right part of the diagram shows the situation for the critical CHR pair which is joinable by the following argument. For $\forall v \in S(\sigma_{CP})$ we know that $tr_{G \Rightarrow G_1}(v)$ and $tr_{G \Rightarrow G_2}(v)$ are defined, thus by Def. 7.2, $v$ is never removed and the corresponding node is still present in $X_1$ and $X_2$. For $\sigma'_1 \equiv_v \sigma'_2$ we similarly observe, that the corresponding node encodings remain. As $v \in S(\sigma_{CP})$ is by Def. 7.2 a global variable, it coincides with the restriction on the bijection between $X_1$ and $X_2$.

Hence, $\forall v \in S(\sigma_{CP})$ their images in $X_1$ and $X_2$, as well as the corresponding encodings in $\sigma'_1$ and $\sigma'_2$ are uniquely determined. Furthermore, all remaining node identifier variables in $\sigma'_1$ and $\sigma'_2$ are local (otherwise a track morphism for only one GTS derivation would exist), hence we transfer the bijection between $X_1$ and $X_2$ over to $\sigma'_1 \equiv_v \sigma'_2$ as in Lemma 18.5.

Therefore, for all overlaps $\sigma_{CP}$ with $G(\sigma_{CP})$ holding we know that the corresponding critical CHR pair is joinable, and hence by Cor. 19.4, that the CHR program is $G$-local-confluent. As it is $G$-terminating as well, it is $G$-confluent.

Lemma 19.5 gives us the following important corollary, which answers our initial question on what it means to the original GTS, when we investigate $G$-confluence of its corresponding GTS-CHR program.

**Corollary 19.7 ($G$-Confluence Implies GTS Confluence).** If a $G$-terminating GTS-CHR program is $G$-confluent, then the corresponding GTS is confluent.

**Proof.** Due to [Plump, 2005], strong joinability is a sufficient criterion for confluence of a terminating GTS. Therefore, this follows directly from Lemma 19.5.

Practically, with Corollary 19.7 we can reuse the automatic confluence check for terminating CHR programs [Abdennadher et al., 1999] to prove confluence of a terminating GTS-CHR program. As Lemma 19.3 showed, it is sufficient to only consider overlaps satisfying the graph invariant $G$. Whenever all the resulting critical CHR pairs are joinable, the CHR program is $G$-confluent according to Corollary 19.4. This, in turn, is sufficient for proving confluence of the original GTS.

**Excursion: Mea Culpa**

In earlier publications [Raiser and Frühwirth, 2009c, 2010], I have included a stronger version of Lemma 19.6, which had a weaker precondition, in that it did not impose the restriction on the track morphology. However, this restriction is necessary.

Consider the GTS given in Figure V.6, which is based on two different node types that we call $a$ and $b$. There is exactly one critical GTS pair, stemming from the first two rules. In one derivation, the edge is simply removed, whereas the other derivation removes the left node, yet reaches an isomorphic final graph by reintroducing a corresponding node. However, the track morphology for the node in question is only defined for the first derivation due to the deletion.

As the following example shows, in the CHR context, the final isomorphism cannot be transferred to equivalence, because of the global variable used to identify the undeleted node.

**Example 19.8.** The relevant overlap state in CHR is

$$\sigma = \langle a(N_1, 1), a(N_2, D), \text{edge}(E, N_1, N_2); \top; \{N_1, N_2, D, E\} \rangle.$$
19. Confluence Analysis

This leads to the critical pair \((\sigma_1, \sigma_2)\) with

\[
\begin{align*}
\sigma_1 &= \langle a(N_1, 0), a(N_2, D'); D' = D - 1; \{N_1, N_2, D}\rangle \\
\sigma_2 &= \langle b(N', 1), a(N_2, D), \text{edge}(E', N', N_2); T ; N_2, D\rangle.
\end{align*}
\]

The remaining possible step then is \(\sigma_2 \rightarrow_e^3 \tau\) = \([\langle a(\tilde{N}, 0), a(N_2, D'); D' = D - 1; \{N_2, D}\rangle]\) and we have \(\sigma_1 \not\equiv_e \tau\), because of the global variable \(N_1\).

19.3 Discussion

In this section, we elaborate on two canonical examples, inspired by [Plump, 2005], that highlight different properties of critical pairs. The first example clarifies, why in the GTS context joinability of critical pairs is not sufficient for local confluence, and hence, why we require strong joinability instead. The second example then discusses a counterexample demonstrating that strong joinability of critical pairs is only a sufficient criterion, not a necessary one.

Example 19.9. Consider the following rules which use two different edge types: \(a\) and \(b\)

\[
\begin{align*}
\text{r1:} & \quad \begin{array}{c}
\begin{array}{c}
\xrightarrow{a}
\end{array}
\end{array} \\
\text{r2:} & \quad \begin{array}{c}
\begin{array}{c}
\xrightarrow{a}
\end{array}
\end{array} \\
\text{r3:} & \quad \begin{array}{c}
\begin{array}{c}
\xrightarrow{a}
\end{array}
\end{array}
\end{align*}
\]

The only critical GTS pair of these rules is joinable. This is possible in the GTS case, because the resulting graphs, shown below, are isomorphic.

However, the track morphisms of the above derivations are incompatible, i.e. the strong joinability condition from Definition 7.2 cannot be satisfied. As the following derivation shows, this hinders monotonicity and joinability is lost, when the critical pair is embedded into a larger context.
The two resulting states are no longer isomorphic and also cannot be joined, as no more rules are applicable to them. Therefore, this GTS is not locally confluent, although all its critical GTS pairs are joinable.

We now examine this scenario in CHR. The two GTS rules then become the following CHR rules:

\[
\begin{align*}
\text{r1} &\quad \text{node}(N_x, D_x), \text{node}(N_y, D_y), a(E, N_x, N_y) \\
&\iff \text{node}(N_x, D_x'), \text{node}(N_y, D_y'), b(E', N_x, N_x), D_x' = D_x + 1 \land D_y' = D_y - 1
\end{align*}
\]

\[
\begin{align*}
\text{r2} &\quad \text{node}(N_x, D_x), \text{node}(N_y, D_y), a(E, N_x, N_y) \\
&\iff \text{node}(N_x, D_x'), \text{node}(N_y, D_y'), b(E', N_y, N_y), D_x' = D_x - 1 \land D_y' = D_y + 1
\end{align*}
\]

We now consider the critical CHR pair corresponding to the above critical GTS pair. It is generated by fully overlapping both rule heads, resulting in the overlap

\[
\sigma_{CP} = \langle \text{node}(N_1, D_1), \text{node}(N_2, D_2), a(E, N_1, N_2); \top; \emptyset \rangle,
\]

with \( \emptyset = \{N_1, N_2, D_1, D_2, E\} \). The resulting critical CHR pair \((\sigma_1, \sigma_2)\) is

\[
\langle \text{node}(N_1, D_1'), \text{node}(N_2, D_2'), b(E', N_1, N_1); D_1' = D_1 + 1 \land D_2' = D_2 - 1; \emptyset \rangle,
\]

\[
\langle \text{node}(N_1, D_1), \text{node}(N_2, D_2), b(E, N_2, N_2); D_1 = D_1 - 1 \land D_2 = D_2 + 1; \emptyset \rangle.
\]

It is clear that \( \sigma_1 \not\equiv \sigma_2 \), because \( CT \not\models (D_1' = D_1 + 1 \land D_2' = D_2 - 1) \rightarrow \exists \emptyset N_1 = N_2 \) as required by Theorem 3.

The strong nodes \( N_1 \) and \( N_2 \), i.e. \( N_1, N_2 \in \emptyset \), enforce compatible track morphisms, and hence are responsible for the non-joinability above. If we instead want to test non-strong joinability, we can do so as well by setting \( \emptyset = \emptyset \). Then, the two states \( \sigma_1 \) and \( \sigma_2 \) are indeed equivalent by Definition 8.2, as \( N_2 \) is existentially quantified and the remaining conditions of Theorem 3 hold as well.

**Example 19.10.** Another example by Plump [2005] is the following GTS, which is terminating and confluent, however, the critical GTS pair from the overlap of rule r1 with itself is not strongly joinable. This is a counterexample that shows strong joinability of critical GTS pairs is only a sufficient criterion for confluence of a terminating GTS.
The GTS works as follows: If there is at least one loop in the graph, then all but a last loop are removed by the first rule. Additionally, all non-loop edges are removed by the second rule. Therefore, the remaining final graph contains zero or one loops and no other edges, and hence the GTS is terminating and confluent due to graph isomorphism. The first rule is encoded in CHR as

\[
\begin{align*}
    r1 @ & \quad \text{node}(N_x, D_x), a(E_x, N_x, N_x) \setminus \text{node}(N_y, D_y), a(E_y, N_y, N_y) \\
    \iff & \quad \text{node}(N_y, D'_y), D'_y = D_y - 2.
\end{align*}
\]

Completely overlapping the rule with itself yields the overlap

\[
\sigma_{CP} = \langle \text{node}(N_1, D_1), \text{node}(N_2, D_2), a(E_1, N_1, N_1), a(E_2, N_2, N_2); \top; \forall \rangle,
\]

with \( V = \{N_1, N_2, D_1, D_2, E_1, E_2\} \) resulting in the critical CHR pair \((\sigma_1, \sigma_2)\) with

\[
\begin{align*}
    \sigma_1 & = \langle \text{node}(N_1, D'_1), \text{node}(N_2, D'_2), a(E_1, N_1, N_1); D'_2 = D_2 - 2; \forall \rangle, \\
    \sigma_2 & = \langle \text{node}(N_1, D'_1), \text{node}(N_2, D'_2), a(E_2, N_2, N_2); D'_1 = D_1 - 2; \forall \rangle.
\end{align*}
\]

Analogously to the previous example, the two states are not equivalent and cannot be joined, therefore the corresponding critical GTS pair is not strongly joinable. Again, setting \( V = \emptyset \) results in both states becoming equivalent. As before, this reflects that for the critical GTS pairs the two corresponding graphs are isomorphic.

As the above example has shown, strong joinability of all critical pairs of a GTS is not a necessary condition for local confluence. Therefore, this analysis method cannot decide confluence of a terminating GTS, in contrast to the situation for CHR programs. This situation is complicated further by the realization that termination of a GTS and its corresponding GTS-CHR program coincide (cf. Cor. 18.9). Hence, for a terminating GTS, confluence is undecidable, yet for its, also terminating, GTS-CHR program confluence and \( G \)-confluence are decidable.

This seeming contradiction is resolved by remembering that the definitions of joinability used in the two systems differ in the involved equivalence relations: For a GTS we require graph isomorphism, whereas a GTS-CHR program requires CHR state equivalence. As we have shown before, state equivalence implies graph isomorphism (cf. Lemma 18.5), but in general not vice versa. For this reason, the GTS presented in the above example is a terminating and confluent GTS, yet its corresponding GTS-CHR program is \( G \)-terminating and not \( G \)-confluent.

In summary, every \( G \)-confluent GTS-CHR program corresponds to a confluent GTS, but there exist confluent graph transformation systems, for which their corresponding GTS-CHR programs are not \( G \)-confluent. This fact can be considered the reason for the difference in decidability of confluence for both systems.

An interesting corollary stems from the following thought experiment: Let us consider a CHR dialect that has no support for global variables, i.e. all variables in states are local. For such a CHR dialect the typical critical pair analysis for confluence becomes insufficient. Furthermore, our results confirm for states that encode graphs, that the state equivalence relation then coincides with graph isomorphism, and therefore, confluence coincides in both systems. This in turn implies that the undecidability of confluence shown by Plump [2005] transfers directly to this CHR dialect.
20 Program Equivalence Analysis

In this section, we introduce our notion of program equivalence for GTSs. It is based on our previous embedding of GTS in CHR and our discussion of invariant-based program equivalence. To the best of our knowledge, a similar input-output approach for terminating graph transformation systems has not yet been investigated.

As a basis, we define the property of $\mathcal{NF}$-equivalence for two graph transformation systems $S_1$ and $S_2$. It is motivated by $\mathcal{NF}$-equivalence in the context of CHR as given in Definition 15.9.

Definition 20.1 ($\mathcal{NF}$-Equivalence). For a typed graph $G$ and a graph transformation system $S$, we call the set

$$\mathcal{NF}_S(G) := \{ H \mid G \Rightarrow^* S H \not\Rightarrow \} \equiv$$

the normal forms (or normal form graphs) of $G$.

A typed graph $G$ is $\mathcal{NF}$-equivalent with respect to $S_1$ and $S_2$ if and only if $\mathcal{NF}_{S_1}(G) = \mathcal{NF}_{S_2}(G)$. If $S_1$ and $S_2$ are clear from the context, we simply write $G$ is $\mathcal{NF}$-equivalent.

In analogy to CHR interfaces, we assume that the two compared graph transformation systems share a common type graph $\mathcal{T}G$, which is a subgraph of both complete type graphs. Furthermore, we assume that all labels that are not in $\mathcal{T}G$ are unique. Hence, when embedding the GTSs in CHR the labels present in the shared type graph $\mathcal{T}G$ correspond to the interface $I$. This in turn allows us to define $\mathcal{NF}$-$\mathcal{T}G$-equivalence for two graph transformation systems.

Definition 20.2 ($\mathcal{NF}$-$\mathcal{T}G$-Equivalence). Let $G$ be a graph and $S_1$ and $S_2$ be graph transformation systems with a shared type graph $\mathcal{T}G$. Then, $G$ is $\mathcal{NF}$-$\mathcal{T}G$-equivalent with respect to $S_1$ and $S_2$ if and only if $\mathcal{NF}_{S_1}(G) = \mathcal{NF}_{S_2}(G)$ and all normal form graphs in $\mathcal{NF}_{S_1}(G)$ and $\mathcal{NF}_{S_2}(G)$ are typed over $\mathcal{T}G$.

In the GTS context, the type graph remains unchanged during derivations. However, we want to ensure that the resulting graphs rely on only certain types that are also available in the other GTS. Therefore, when comparing the normal forms, we require that these graphs only contain typed nodes and edges associated with the shared type graph $\mathcal{T}G$.

We can now finally define our notion of program equivalence for graph transformation systems.

Definition 20.3 (GTS Program Equivalence). Let $S_1$ and $S_2$ be two graph transformation systems with a shared type graph $\mathcal{T}G$ and let $\mathcal{G}$ be the graph invariant. $S_1$ and $S_2$ are $\mathcal{T}G$-equivalent if and only if for all graphs $G$ typed over $\mathcal{T}G$ it holds that $G$ is $\mathcal{NF}$-$\mathcal{T}G$-equivalent with respect to $S_1$ and $S_2$.

For the following theorem, we conclude our underlying idea: In order to compare program equivalence of two graph transformation systems, we first encode them into CHR and then apply the CHR program equivalence test. This yields the following sufficient criterion for GTS program equivalence.

Theorem 12 (CHR Program Equivalence Implies GTS Program Equivalence). Let $S_1$ and $S_2$ be graph transformation systems with a shared type graph $\mathcal{T}G$ and $P_1$ and $P_2$ their corresponding GTS-CHR programs with interface $\mathcal{I}$. $S_1$ and $S_2$ are $\mathcal{T}G$-equivalent if $P_1$ and $P_2$ are $\mathcal{G}$-$\mathcal{I}$-equivalent.
20. Program Equivalence Analysis

Proof. Let \( G \) be a graph typed over \( TG \). Then the state \( \sigma = \langle \text{chr(ground, } G); \top; \emptyset \rangle \) is a \( \Pi \)-state and \( \mathcal{NF}\Pi\)-equivalent with respect to \( P_1 \) and \( P_2 \). Therefore, for each final state \( [\tau] \in \mathcal{NF}_{P_1}(\sigma) \) (w.l.o.g. \( P_1 \), as \( \mathcal{NF}_{P_1}(\sigma) = \mathcal{NF}_{P_2}(\sigma) \)) we have \( [\sigma] \rightarrow^\ast_{P_1} [\tau] \) and \( [\sigma] \rightarrow^\ast_{P_2} [\tau] \) and \( \tau \) is a \( \Pi \)-state. By Thm. 11 we know that there exist corresponding derivations \( G \Rightarrow^\ast S_1 G_1 \) and \( G \Rightarrow^\ast S_2 G_2 \), such that \( \tau \) is a \( G \)-state based on \( G_1 \) and a \( G \)-state based on \( G_2 \), or by Lemma 18.5, that \( G_1 \simeq G_2 \). Due to Thm. 11, \( G_1 \) and \( G_2 \) are final states w.r.t. \( S_1 \) and \( S_2 \). As \( \tau \) is a \( \Pi \)-state, we know both graphs \( G_1 \) and \( G_2 \) contain only nodes from \( TG \) (which corresponds to \( \Pi \)). Hence, \([G_i] \in \mathcal{NF}_{S_i}(G)\) and \([G_i] \in \mathcal{NF}_{S_i}(G)\) for \( i = 1, 2 \), where \([H]\) denotes the equivalence class of a graph \( H \) with respect to graph isomorphism \( \simeq \).

Finally, there cannot exist an additional \( [G'] \in \mathcal{NF}_{S_i}(G) \) that was not covered by the above, because Thm. 11 guarantees that there exists a CHR derivation leading to a corresponding final state \( [\tau] \). Therefore, we have that \( \mathcal{NF}_{S_1}(G) = \mathcal{NF}_{S_2}(G) \), i.e. \( G \) is \( \mathcal{NF}_{TG} \)-equivalent, and hence, \( S_1 \) and \( S_2 \) are \( TG \)-equivalent.

The reverse direction of Thm. 12 cannot be proved in a similarly simple way. Analogously, to confluence the problem stems from the irreversibility of Lemma 18.5, which relates graph isomorphism to CHR state equivalence.

20.1 Redundant Rule Removal

The redundant rule removal algorithm is an application of program equivalence presented in Abdennadher and Frühwirth [2003]. It can be applied to graph transformation systems embedded in CHR. It requires the CHR program to be \( G \)-terminating and \( G \)-confluent, which implies that the GTS is required to be confluent and terminating as well. Any redundant rule of such a program then corresponds to a redundant rule for derivations of the GTS.

The basic idea of the algorithm is to try to remove a rule from the program and compare this modified program with the original one. If both are equivalent the rule is redundant and can be removed. Depending on the order in which rules are tried, different results are possible and this algorithm, hence does not guarantee to yield an equivalent program with the minimal number of rules possible. Nevertheless, it proved to be an important algorithm especially in the context of automatically generated programs [Abdennadher and Sobhi, 2008, Raiser, 2008].

Example 20.4. Consider the GTS given in Figure V.7. The two rules replace edges of type \( a \) by edges of type \( b \). An application of our program equivalence results is the automatic removal of the second rule due to its redundancy. The following is the encoding of this GTS in CHR.
The algorithm tries to remove the second rule from the program and then compare the two programs according to the program equivalence test. To this end, when the second rule state is investigated, the two computations may use different rules, but both programs compute a final state consisting of a graph with three nodes, a b-edge from the first to the third node, and a b-edge from the first to the second node.

The following example demonstrates that global variables in CHR play an important role, similarly to confluence analysis. Directly formulating our approach in the GTS context, hence, may require a similar track morphism-based restriction to the graph isomorphism used to compare the final results (cf. Section 21).

**Example 20.5.** Consider the two graph transformation systems shown in Figure V.8. Using the graph on the left-hand sides of the two rules as input to both GTSs leads to isomorphic results as shown in Figure V.9 (1). However, assuming the slightly extended graph, shown in Figure V.9 (2), instead yields two non-isomorphic result graphs.

Therefore, only examining the critical states within the GTS using non-strong derivations is insufficient. Applying our approach to the GTS in Figure V.8, however, gives the intended result: non-equivalent final states.

### 21 Related and Future Work

The work in this chapter clearly relates to the lingua-franca argument, and hence, to all existing encodings of rule-based approaches in CHR (cf. Section 5.3). However, as stated before, there is little prior research into embeddings of graph-based formalisms.
Our encoding of GTSs in CHR is but a first foray into this field and there exist several important extensions for GTSs, which can be investigated for it. Firstly, our work should be extended to typed attributed GTSs. An attributed graph associates nodes and edges to attributes stemming from a sigma algebra. In other words, it allows assigning values to graph elements and the corresponding operations defined in the sigma algebra can be exploited in rule applications.

Attributed graphs are especially convenient for model driven architecture. They support, for example, assignments of names to classes or cardinalities to associations. Hence, applying our work to the MDA field should be preceded by an extension that covers typed and attributed graphs. Fortunately, CHR provides us with a built-in constraint theory that can be exploited for such an extension. The core idea for achieving this would be the integration of the underlying sigma algebra into the built-in constraint theory.

Venturing further into the MDA field, one could envision CHR as a basis for model transformations. A particularly interesting topic in this field is the Object Constraint Language (OCL) [OCL]. We can easily envision the power of the lingua-franca argument here: the encoding of attributed GTSs in CHR and CHR’s historic background in constraint programming, proposes a possible unification of the involved techniques. With a single CHR program one could model transformations and their OCL constraints, execute them, and provide them with an underlying linear-logical semantics.

Another important extension of GTSs are application conditions, which restrict applicability of rules. So called negative application conditions (NACs) specify a graph, for which no isomorphic subgraph must be present in the host graph if the rule is to be applied. In concept, this is similar to guards in CHR, but lacks their locality. Whereas a CHR guard only poses conditions onto constraints given in the head, a negative application condition poses such a condition onto the whole host graph.

Negative application conditions cannot be modeled as guards in CHR, because that would require the guards to argue over CHR constraints. Supporting NACs in our encoding will most likely involve some scheduling of rules, such that one rule first determines whether the NAC is violated, and the actual rule is only applicable if that is not the case. It is important to ensure, that the NAC is checked before each rule application, because a corresponding subgraph could be created by any rule application, causing a violation of the NAC.

Based on the lingua-franca argument, our encoding can serve as a foundation for transferring results between these two formalisms. We have shown this exemplarily for confluence and program equivalence, but the principle applies just as well to other analysis methods. Termination analysis, in particular, might be worthwhile to look into due to the direct correspondence of termination of a GTS and its GTS-CHR program (cf. Corollary 18.9).

Finally, a recent result by Plump [2010] has identified an extension of graph transformation systems called coverable hypergraph-transformation system. This extension is interesting, as confluence of such a terminating system is decidable, based on extending critical pairs with a non-deletable context. This is remarkably similar to the behavior of global variables in our encoding of critical pairs in CHR. Although none of our works is cited in [Plump, 2010], we assume that they, and our personal communications with Plump, partly inspired the idea of coverable GTSs. However, a closer comparison is left for future work.
Chapter VI

Conclusion

We can only see a short distance ahead, but we can see plenty there that needs to be done.
— Alan Mathison Turing (1912–1954), Computer Scientist

In this section, we summarize this thesis, which is based on the idea of CHR as a lingua-franca for rule- and logic-based formalisms. In this respect, we discovered several shortcomings of the current formulations of the operational semantics of CHR, while experimenting with an encoding of graph transformation systems in CHR. Therefore, we planned to examine these shortcomings, improve our encoding of GTSs in CHR, and return to the lingua-franca argument by comparing analysis methods for confluence and program equivalence available in both systems. The following sections respectively conclude these topics: equivalence-based operational semantics, program analysis methods, and encoding of GTSs.

Despite the above quote, we refrain from a discussion of future work here, as it has already been extensively laid out in Sections 12, 16, and 21.

22 Equivalence-based operational semantics

In Chapter III, we addressed the aforementioned shortcomings and proposed a new formulation for an equivalence-based operational semantics $\omega_e$. We have given an axiomatic definition of state equivalence in CHR that forms the foundation of $\omega_e$, and provided a sufficient and necessary criterion for deciding it. Based on its compliance with rule applications, we introduced a novel view of the operational semantics as a rewriting system on equivalence classes of CHR states.

We have observed that all prior operational semantics focused on either being analytical or implementation-oriented, which in particular, became manifest in their treatment of propagation rules. Our introduction of the operational semantics $\omega$ fills this gap, as it is complete with respect to the intended abstract semantics of CHR, and furthermore provides a terminating execution model for propagation rules.

We have adapted our previous work from $\omega_e$ to $\omega$, such that it also provides an axiomatic state equivalence definition $\equiv_1$, a decision criterion for $\equiv_1$, and an equivalence-class based formulation of the operational semantics.

Furthermore, we have formalized merging of two CHR states via the merge operator $\diamond$, which was also lifted to equivalence classes. We identified several important properties of this
operator, in particular, that the set of equivalence classes of states together with \( \diamond \) forms a commutative monoid. We could exploit this to define a partial order on CHR states, which facilitates program analysis, as we demonstrated in Chapter IV.

Finally, we investigated different formulation possibilities for our proposed operational semantics, identified a differing termination behavior for \( \omega! \) and investigated its expressivity. From the analysis of expressivity of \( \omega! \), we could derive a source-to-source implementation to support our argument that \( \omega! \) indeed fills the aforementioned gap.

In summary, our equivalence-based operational semantics provides several advantages over other existing formulations: declarative states, a transition system where each transition corresponds directly to a rule application, a solid state equivalence definition – including equivalence classes, which abstract over possible syntactic variations of a state. It is also the first formulation of the operational semantics, which corresponds closely to the traditional informal formulations of CHR derivations. As Example 8.8 has shown, even the most simple CHR queries yield complicated final states under existing operational semantics. Except for the required syntax of CHR states, Example 8.10 demonstrates that with our formulation we can write down derivations on a very intuitive level, while still remaining true to the formal semantics.

### 23 Program Analysis

Next, in Chapter IV we generalized an idea by Duck et al. [2007]. We investigated the usage of invariants in CHR program analysis, in particular, we discussed the application of our partial order on CHR states for the definition of minimal elements. Although the core idea for this has already been provided in [Duck et al., 2007], we have identified multiple shortcomings of their definitions of a partial order and minimal elements.

Hence, after investigating the general effect of adding invariants to program analyses, we reproduced the results on invariant-based confluence by Duck et al. [2007] in our equivalence-based framework.

Furthermore, we identified possible improvements for the program equivalence results by Abdennadher and Frühwirth [1999], namely, support for non-terminating and non-confluent programs, as well as program interfaces. We additionally extended their results to yield a sufficient and necessary invariant-based program equivalence criterion.

Therefore, our work makes these program analysis methods applicable to a strictly larger set of CHR programs. The inclusion of invariants ensures that implicit assumptions are accounted for, as they need to be made explicit in order to successfully apply our results.

Formulations of these methods, that existed in the CHR literature before, typically fail for programs that involve common structures like numbers, lists, linked lists, or graphs, due to testing inconsistent states. In contrast, our improved methods ensure that all tested states contain consistent data structures, which makes failed tests on such states significantly more meaningful.

### 24 Encoding of Graph Transformation Systems

Finally, in Chapter V we revisited our encoding of graph transformation systems. We found the equivalence-based operational semantics to be more elegant in analyzing the properties of the encoding and to lead to more concise proofs.
We proved soundness and completeness of the encoding, and further investigated its properties, for which we applied the idea of invariants. We identified an invariant that describes graph states and analyzed the correspondence between graph isomorphism and state equivalence of graph states. Furthermore, we found that the CHR encoding supports derivations on partial graphs, which would not be possible in an ordinary GTS.

Our work then culminated in the investigation of confluence and program equivalence of encoded GTSs, which brought together all previous results: the encoding of GTSs in CHR, our results on invariant-based program analysis, and the equivalence-based operational semantics as the formal foundation. We have shown that invariant-based confluence analysis of an encoded terminating GTS is a sufficient criterion for its confluence and gave the first program equivalence result for GTSs, which we successfully applied for redundant rule removal. We can therefore conclude, that this work confirms the suitability of CHR as a lingua-franca.

We found our equivalence-based operational semantics beneficial for this line of research and successfully demonstrated the inherent potential for cross-fertilization.

---

**Excursion: My Contributions**

I have already listed previously published work contained in this thesis in Section 4. In this excursion, I discuss my individual contributions to this thesis, especially with regard to co-authored publications.

To begin with, Prof. Dr. Frühwirth has served as a supervisor for this thesis, and hence, accompanied the complete work. He initially suggested the comparison of graph transformation systems with CHR and provided valuable feedback on how to improve publications for submission. Nevertheless, all works cited in Section 4 that have been authored by myself and Prof. Dr. Frühwirth, contain the results of my own work. In particular, this includes all contents of Chapters IV and V, except for the duly marked contributions of third parties.

Chapter III is founded on the fruitful joint work with Hariolf Betz. In particular, Sections 8, 9, and 11 have previously appeared in joint peer-reviewed publications. My approach on the semantics proposed in this chapter was focused on operational behaviors due to the GTS context of this work. Contrarily, Hariolf Betz investigates the linear-logical semantics of CHR in his work, and hence, contributed corresponding elements.

The core idea of adding persistent constraints to CHR cannot be clearly attributed to a single author. However, Hariolf Betz provided the axiomatic state equivalence definition for \( \omega_e \) and \( \omega_i \), proved their compliance with rule applications, and their soundness and completeness with respect to \( \omega_{va} \).

In contrast, my own contributions include the criteria for deciding state equivalence (Theorems 3 and 5). Furthermore, I recognized the potential, rule compliance of \( \omega_e \) and \( \omega_i \) has for a simplified formulation of the operational semantics. Based on this, I introduced the point of view of CHR as a rewriting system of equivalence classes, leading in particular to all results on the merge operator \( \diamond \), given in Section 10.

Finally, I discovered the differing termination behavior of \( \omega_i \) and \( \omega_l \) (cf. Section 11.3) and analyzed expressivity of \( \omega_i \). The latter resulted in an implementation of \( \omega_i \) in the form of a source-to-source transformation (cf. Sections 11.4 and 11.5).
24. Encoding of Graph Transformation Systems
Bibliography


Bibliography


CULP09, 2009. Commercial Users of Logic Programming Workshop, Pasadena, CA, USA.


Chapter VI


Drools. JBoss Drools. URL http://jboss.org/drools.


François Fages, Cleyton Mário de Oliveira Rodrigues, and Thierry Martinez. Modular CHR with *ask* and *tell*. In Schrijvers et al. [2008], pages 95–110.


RulesFest, 2010. International Conference on Reasoning Technologies, San Jose, CA, USA.


Bibliography


Dean Voets, Paolo Pilozzi, and Danny De Schreye. A New Approach to Termination Analysis of Constraint Handling Rules. In Djelloul et al. [2007], pages 77–89.


Pieter Wuille, Tom Schrijvers, and Bart Demoen. CCHR: the fastest CHR Implementation, in C. In Djelloul et al. [2007], pages 123–137.
Index

$\mathcal{NF}$-Equivalence, 93, 123

Acceptable Encoding, 61
Aggregates, 104
Answer, 61

Behavioral Equivalence, 30
Bisimilarity, 29

Commutative Monoid, 51
Completeness, 110
Completion, 24
Complexity, 24, 25
Computation, see Derivation
Computational Power, 25
Configuration, see State
Confluence, 26, 81
  Invariant-based, 85
  Local, 27, 82, 86
  Test, 83, 86, 119
Constants, 7
Constraint, 8
  Built-in, 8
  CHR, see Constraint
  Identified, 11
Constraint Theory, 8
Context Graph, 21
Critical Pair, 26, 82, 114
  CHR, 82
  GTS, 28

Dangling Edge, 21, 102
Degree, 19
Degree Variable, 100
Derivation, 10
  Finite, 10
  Infinite, 10
  Length, 10
  Step, 10
Deterministic, 10
Double-Pushout Approach, 21

Encoding
  Rule, 101
  Type Graph, 100
  Typed Graph, 100
Equivalence Relation, 36, 46

Global Variables, 34, 46
Gluing Condition, 21
Goal, 33
Graph, 19
Graph Invariant, 106
Graph Isomorphism, 20, 108
Graph Morphism, 19
  Inclusion, 20
  Injective, 20
  Surjective, 20
Graph Production Rule, 20
Graph Transformation System, 20
Ground, 7
GTS-CHR Program, 101

Host Graph, 20

Instance, 7
Interface, 92
Invariant, 75
Invariant-based Confluence, 85

Joinable, 28, 81, 89
  Strongly, 28

Linear Constraints, 45
Linear Store, 46
Logical Reading
- First-Order Rules, 16
- States, 17
- Linear-Logic, 17

Macro Step, 94
Match Morphism, 21
Merge Operator, 50
Minimal Elements, 80, 116
Modularity, 24, 25
Monotonicity, 51

n-Equivalence, 29
Newman’s Lemma, 82
Non-Deterministic, see Deterministic Normal Form, 93, 123

Observable Confluence, 27, 81
Operational Equivalence, 30, 89
Operational Semantics
- Equivalence-based, 41
- Persistent, 47
- Refined, 13
- Rule Priorities, 13
- Theoretical, 11
- Very Abstract, 10

Overlap, 82

Partial Order, 53
Persistent Constraints, 45
Persistent Store, 46
Priorities
- Dynamic, 14
- Static, 14

Production Rule, see Graph Production Rule
Program, 8
Propagation History, 12

Range-Restricted, 8, 55
Reachability Relation, 10
Rule Graph, 20
Rules, 8
- Body, 8
- Guard, 8
- Head, 8
- Identifier, 8
- Pathological, 62

Propagation, 8
Range-Restricted, 8
Simpagation, 8
Simplification, 8

Single-Pushout Approach, 23
Soundness, 110

State, 10
- Critical State, 78
- Final State, 10
- Initial State, 10
- Minimal State, 78
- Overlap, 82
- Rule State, 78

State Splitting, 79
State Transition System, 10
- $\omega_e$, 41
- $\omega_p$, 14
- $\omega_t$, 12
- $\omega_v$, 47
- $\omega_e$, 44
- $\omega_{set}$, 15, 72
- $\omega_{eva}$, 10

States
- $\omega_l$-State, 46
- $\omega_r$-State, 33
- $\omega_{pr}$-State, 14
- $\omega_t$-State, 11
- $\omega_{va}$-State, 10

Strong Joinability, 28, 117
Strong Nodes, 109
Subgraph, 20
Substitution, 7, 36, 46
Symbols
- Constraint
- Built-in, 8
- CHR, 8
- Function, 7
- Predicate, 7

Term, 7
- Function, 7
- Termination, 24, 25, 59, 77, 113
- $\mathcal{I}$-Termination, 77
Track Morphism, 23
Transition Relation, 10
Trivial Non-Termination, 11
Type Graph, 19
Typed Graph, 20

Variable Renaming, 37
Variables, 7
  Global, 8
  Local, 8
  Strictly local, 8
Variant, 7