THE UNIVERSITY OF
MELBOURNE

Compilation of Constraint Handling Rules

Gregory J. Duck

December 2005

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia

Abstract

Constraint Handling Rules (CHRs) are a committed choice rule based programming language designed for the implementation of incremental constraint solvers, or for extending the functionality of existing solvers. A CHR program is a set of rules, which are exhaustively applied to some input constraints, until no further rule is applicable.

CHRs have become increasingly popular in recent years, mainly because CHRs are a very high-level language, which makes it possible to specify complex problems with just a few rules.

Despite the popularity of CHRs, most CHR compilers or interpreters are still very simple. For instance, very few implementations attempt any meaningful optimisation, which results in poor performance at runtime. The aim of this thesis is to modernise the compilation of CHRs, and therefore make it a far more practical programming language.

We take both a theoretical and practical approach to compiling CHRs. For the theoretical part, we formalise the operational semantics of CHRs as used by most current CHR implementations. This helps us verify some important theoretical results for CHRs, and forms the basis for CHR program analysis, which is an essential part of CHR compilation.

We look at optimising CHRs based on various information available to the compiler. Some of the optimisations depend on the results of the program analysis. We evaluate the optimisations based on some experimental results.

This thesis is also a handbook on CHR compilation, with all of the essential information contained within. Most of the ideas in this thesis could easily be adapted to other CHR compilers in other environments.

Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Signed,

Gregory J. Duck
21st December 2005

Acknowledgements

Financial support for this thesis was provided by the federal government of the Commonwealth of Australia in the form of a Australian Postgraduate Award (APA), and by the state government of Victoria in the form of an Information Communication Technology (ICT) scholarship.

I would especially like to thank my supervisor, professor Peter J. Stuckey, for his (generally) excellent supervision.

I would also like to thank the following people. Maria Garcia de la Banda for her help and guidance with respect to the HAL compiler, and all of its idiosyncrasies. Christian Holzbaur for his help with some of the practical aspects of building a CHR compiler. Tom Schrijvers and Jeremy Wazny for many interesting (and relevant) discussions about CHRs. For their help with the HAL compiler, I'd also like to thank two of its past developers, David Overton and Ralph Becket.

I'd like to especially thank Martin Sulzmann for his help and support, and for helping me escape Melbourne's cold cold winters.

And a special thanks to Priscilla, for all of her love and patience over the years.

Preface

This thesis consists of 9 chapters, including introduction and conclusion. Chapter 2 presents the necessary background information in order to understand this thesis. Chapter 3 formalises the operational semantics of CHRs used by modern CHR compilers. Chapter 4 details a simple compilation strategy for CHRs. Chapter 5 presents an analysis framework for CHRs, and includes two instances of that framework. Chapter 6 presents a confluence test based on the results of the analysis. Chapter 7 presents several CHR specific optimisations, also based on the analysis. Chapter 8 examines more closely the problem of compiling CHRs which extend constraints solvers. Finally, in Chapter 9, we conclude.

Portions of this thesis are based on joint work, which we list here. Chapter 3 is based on the first part of Duck, Garcia de la Banda, Stuckey and Holzbaur [27]. Section 4.5 of Chapter 4 is loosely based on Duck, Garcia de la Banda and Stuckey [24]. Chapter 5, except for Section 5.6, is based on Schrijvers, Stuckey and Duck [73]. Chapter 6, like Chapter 3, is also based on [27]. Chapter 7 is based on Holzbaur, Garcia de la Banda, Stuckey and Duck [45]. Finally, Chapter 8 is based on Duck, Stuckey, Garcia de la Banda and Holzbaur [26]. The rest of the thesis is entirely my own work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Constraint Handling Rules

Constraint Handling Rules (CHRs) are a rule based programming language designed for the implementation of incremental constraint solvers, or for extending the functionality of existing solvers. The popularity of CHRs has significantly increased in recent years. This is principally because, given their high-level and declarative nature, many problems can be concisely expressed in CHRs. For example, some applications of CHRs include

- agent reasoning in the FLUX [84] language;
- type inference in the Chameleon [81] system; and
- many constraint logic programming applications such as the Munich Rent Advisor (MRA) [34] or university course time tabling with soft constraints [2].

More recently, CHRs are being used as a general rewrite language, rather than a language exclusively for writing constraint solvers.

CHRs are part of the rule based programming paradigm. A program consists of a sequence of rules, and these rules are exhaustively applied to some initial input until no more rule application is possible. The result is the output of the program. There are many other programming languages in existence, including reduction systems, production systems, etc., that fit under the rule based paradigm.

Rules in CHRs define relations between constraints. Three types of rules are supported: rules for *rewriting* constraints into other constraints, rules for *adding* new constraints, and rules that are a hybrid of the two. All of these rules consist of a *head* and a *body* separated by an arrow. The rule is applicable if there exist a set of constraints that *match*[1] the head. When the rule is applied, the constraints in the body are added to the goal.

[1] As opposed to unify, as with logic programming languages.

Example 1 *By far the most famous example of a CHRs is the* `leq` *program, which defines a less-than-or-equal-to constraint* `leq(X,Y)` *(i.e.* $X \leq Y$*) as follows.*

```
reflexivity   @ leq(X,X)             <=> true.
antisymmetry  @ leq(X,Y), leq(Y,X)   <=> X = Y.
idempotence   @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity  @ leq(X,Y), leq(Y,Z)   ==> leq(X,Z).
```

This program consists of four rules, one for each line. Each rule has a name*, e.g.* `reflexivity`*,* `idempotence`*, etc., appearing before the '*`@`*' token, which describes the purpose of the rule. The name is optional, and purely syntactic, i.e. doesn't affect the behaviour of the program.*

The `reflexivity` *rule declares that a constraint of the form* `leq(A,A)` *(i.e. the first and second arguments are identical), can be rewritten to the* `true` *constraint. The* `true` *constraint represents the empty or trivial constraint, i.e. contains no information. This rule maintains the identity*

$$\forall X(X \leq X \leftrightarrow true)$$

The `antisymmetry` *rule states that two symmetric* `leq` *constraints, i.e. of the form* `leq(A,B)` *and* `leq(B,A)`*, can be replaced with the equation* $A = B$ *unifying* A *and* B*. This rule maintains the identity*

$$\forall X \forall Y(X \leq Y \wedge Y \leq X \leftrightarrow X = Y)$$

The `idempotence` *rule is more complicated, because there are two* `leq` *constraints appearing in the head of the rule. Notice that the two* `leq` *constraints are identical. The rule states that given two identical* `leq` *constraints,* one *of the constraints (the one matching the right-hand-side of the token '\') can be rewritten to the* `true` *constraint. The purpose of this rule is remove redundant copies of* `leq` *constraints.*

Finally, the `transitivity` *rule states that given two* `leq` *constraints of the form* `leq(A,B)` *and* `leq(B,C)`*, we can* add *the new constraint* `leq(A,C)`*. Unlike the other rules, this rule does not delete any constraint, and it maintains the identity*

$$\forall X \forall Y \forall Z(X \leq Y \wedge Y \leq Z \rightarrow X \leq Z)$$

□

Notice the close relationship between the rules and the mathematical relationship they represent.

A rule with the '`==>`' arrow is a *propagation rule*, which adds the body of the rule. For example, `transitivity` is a propagation rule. A rule with the '`<=>`' arrow, and without the token '\' in the head, is a *simplification rule*, which replaces constraints matching the head with the body. For example, both `reflexivity` and the `antisymmetry` are simplification rules. A rule with the '`<=>`' arrow, but

with a '\' in the head, is a *simpagation rule*, which replaces the constraint matching the right-hand-side of the '\' token with the body. For example, `idempotence` is a simpagation rule.

The input to a CHR program is called the *goal*.

Example 2 *Consider the following goal for the* `leq` *program in Example 1.*

`leq(`A`,`B`), leq(`C`,`A`), leq(`B`,`C`)`

We can apply the `transitivity` *rule on the second and first constraints to add the new constraint* `leq(`C`,`B`)`*.*

`leq(`A`,`B`), leq(`C`,`A`), leq(`B`,`C`), leq(`C`,`B`)`

We can apply the `antisymmetry` *rule to replace the third and fourth constraints with the equation* $B = C$*.*

`leq(`A`,`B`), leq(`C`,`A`),` $B = C$

Now that B *and* C *are equal, we can apply the* `antisymmetry` *rule to replace the first and second constraints with the equation* $A = B$*.*

$A = B$`,` $B = C$

Since no more rules can be applied (since there are no more `leq` *constraints), this is the output of the program.* □

CHRs are a *committed choice* language, which means that if multiple rule applications are possible, then we only try one of them. In Example 2, other combinations of rule applications were possible. If the output state is independent of the combination chosen for any input, then we say that the program is *confluent*. The `leq` program is known to be confluent. In general, confluence is an important property for CHRs (and other rule based languages).

1.2 Aims of this Thesis

Whilst there have been many implementations of CHRs since their invention in 1991, most of these were relatively naive and ad hoc. As the popularity of CHRs has significantly grown in recent years, the need to have better compilers has also grown. The purpose of this thesis is to significantly improve and modernise the compilation of CHRs, which will benefit both CHR programmers and CHR compiler writers.

We divide the aims of this thesis into two main parts.

Formalise the operational semantics of CHRs

The operational semantics of CHRs have long been formalised per se, however most implementations implicitly define a more restrictive operational semantics, where some of the inherent nondeterminism of CHRs (e.g. the choice of rule application) has been removed. In practice, rules are chosen in a textual top-down order, and (sub)goals are executed left-to-right. This is analogous to the execution order which the Prolog [50] programming language uses. Many Prolog applications rely on the execution order, and similarly, many CHR applications have a similar dependency.

The implicit operational semantics for CHRs has never been formalised previous to our work (as far as we are aware). By formalising the semantics, we give the compilation of CHRs a theoretical basis, which can be used to establish correctness, for program analysis, and for optimisation. It also serves as a formal specification for future compiler writers. We can also understand confluence under the implicit semantics, and use it to detect bugs in CHR code.

Improve CHR compilation

Before work on this thesis begun, CHR compilers were relatively simple, and made little or no optimisation. The penalty for using CHRs over a more low-level language could be significant depending on the type of the application. Our aim is to reduce this penalty, and make CHRs a practical option for implementing some applications.

In this thesis we focus on a CHR compiler for the HAL programming language [20, 17]. HAL is a recent Constraint Logic Programming (CLP) language, which makes an ideal host environment for the CHR language. CHR rules can be embedded in HAL programs, and the HAL compiler converts the rules into optimised HAL code.[2] This compiler was implemented by the author of this thesis, and incorporates several kinds of optimisations designed to improve runtime performance. We provide benchmarks to backup the claims of improvements over earlier (and more primitive) compilation schemes.

1.3 A Guide to this Thesis

In Chapter 2 we present all of the necessary background information required in order to understand this thesis. This includes overviews of constraint logic programming, the HAL programming language and CHRs themselves.

Chapter 3 presents a formalisation of the operational semantics of CHRs used in most CHR compilers, including the HAL CHR compiler. This is the implicit semantics identified in Section 1.2. We compare and prove correctness for the new operational semantics with respect to the original version. We show that

[2] It is usually the case that CHRs are compiled into the host language.

several main results in CHRs, e.g. results for confluence and termination, can be lifted to the new semantics.

Chapter 4 presents a basic compiler and runtime environment for CHRs based on the operational semantics from Chapter 3. The basic compiler is no more advanced than existing implementations. It will form the basis for more advanced optimising compilation presented later in this thesis.

Chapter 5 presents an analysis framework of CHRs. This is also based on the operational semantics of Chapter 3. The analysis framework is modelled on abstract interpretation, and there are some unique issues specific to dealing with a language like CHRs. We present two instances of our framework which discover useful information from CHR programs. The information will be used for confluence testing and optimisation presented later in the thesis.

Chapter 6 presents a static check that can partially verify the property of confluence assuming the operational semantics of Chapter 3. Confluence is an important property, and a nonconfluent program is generally considered a buggy program. We use the test to help verify confluence for three large CHR applications.

Chapter 7 presents several optimisations which can improve the resulting executable compared with the basic compilation and other CHR implementations. Some of the optimisations are fairly simple, whilst others rely on the results from program analysis as presented in Chapter 5. We present experimental results to show the benefit of using the optimisations on several sample CHR programs.

Chapter 8 examines the unique issues that arise from CHRs extending another constraint solver. Example 1 is an example of CHRs extending another solver, since the `leq` constraint "extends" a Herbrand solver, which provides the equality constraint. This chapter presents some additional optimisations that were not covered in Chapter 7. It also describes the interface between CHRs and other constraint solvers. We give benchmarks to evaluate the benefits of our approaches.

Finally, in Chapter 9 we conclude.

Chapter 2

Background

2.1 Introduction

In this chapter we cover the necessary background information required for comprehending the rest of this thesis.

Section 2.2 covers the necessary preliminaries, such as an overview of the relevant notation, predicate logic, constraint/logic programming and the HAL programming language. Section 2.3 cover CHRs, including CHR preliminaries, confluence/termination results, the history of CHR compilers to date and related systems.

2.2 Preliminaries

2.2.1 Notation

Most of this notation is standard, so we only give a brief overview.

We use the following notation for the standard logical connectives: $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (iff), $\neg$ (negation), $\models$ (models), $\forall$ (universal quantification) and $\exists$ (existential quantification).

We use standard notation for sets: $\in$ (element of), $\subset$ (strictly contains), $\subseteq$ (contains), $\cup$ (union), $\cap$ (intersection) and $-$ (difference). We use $\emptyset$ to denote an empty set. We can define a set by a listing of its elements enclosed in curly braces, i.e. $\{x_1, ..., x_n\}$ is a set containing elements $x_1, ..., x_n$. Sets may also be defined using *set comprehension*, e.g. $\{x \mid p(x)\}$ defines the set of all x that satisfies property p. The *cardinality* of a set S is given by $|S|$. For finite sets, the cardinality is the number of elements in S. We do not consider the cardinality of infinite sets in this thesis.

We use exactly the same notation for multisets, but introduce the following additional operations: $\uplus$ (multiset union) and $\sqcap$ (multiset intersection). For example, $\{1, 2\} \uplus \{1, 1, 3\} = \{1, 1, 1, 2, 3\}$.

For sequences, we use a notation based on the syntax for the Prolog programming language (which we cover in Section 2.2.4). We use [] to denote an empty

sequence, and $[H|T]$ to denote a sequence with H as the first element (the *head*), and T as the remainder of sequence with H removed (the *tail*). We use operator $++$ to represent sequence concatenation.

The remainder of the notation will be introduced when required.

2.2.2 Predicate Logic

Both constraint and logic programming have their formal basis in *first order predicate logic*, which we briefly describe here. For a more detailed introduction, see [62].

We define $Vars$ to be a set of *logic variables*, $PredSym$ to be a set of *predicate symbols* and $FuncSym$ to be a set of *function symbols* or *functors*. Informally, the variables represent unknowns, predicates are relations which are *true* or *false*, and functions are used to construct values which can be assigned to the variables. We usually use an upper case letter to indicate a variable.

A *term* is constructed from variables and function symbols as follows. A *term* is either

1. a variable $V \in Vars$; or

2. a function symbol $f \in FuncSym$, followed by a comma separated list of terms surrounded by parentheses, i.e. $f(t_1, ..., t_n)$.

We define n to be the *arity* of the term. Sometimes we use f/n group a functor f together with arity n. A term with arity 0 is called a *constant*, in which case we omit parentheses. Terms are *ground* if they contain no variables, otherwise they are *nonground*. We also define a ground term to be a *value*.

An *atomic formula* is of the form $p(t_1, ..., t_n)$ where $p \in PredSym$ is a predicate symbol and $t_1, ..., t_n$ are terms. We also extend the definitions of *arity*, *arguments* and *ground* to atomic formulae.

A *formula* is constructed from atomic formulae using the logical connectives ($\wedge$, $\vee$, $\rightarrow$ and $\neg$) and quantifiers ($\forall V$ and $\exists V$ for $V \in Vars$).

We use a shorthand notation for existential quantification over many variables. Let $vars(A)$ return the variables occurring in any syntactic object A. We use $\exists_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \ldots X_n\} = vars(A)$. Similarly, we use $\bar{\exists}_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \ldots X_n\} = vars(F) - vars(A)$. We also use similar notation with $\forall$ replacing $\exists$.

A *substitution* θ is a mapping from variables to terms represented as

$$\theta = \{X_1/t_1, ..., X_n/t_n\}$$

where $X_1, ..., X_n$ are distinct variables, and $t_1, ..., t_n$ are terms. If A is a syntactic object, then $\theta(A)$ is the syntactic object obtained by replacing each instance of variable X_i with the term t_i. Often we treat a substitution as a conjunction of equations, i.e. $\theta = (X_1 = t_1 \wedge ... \wedge X_n = t_n)$.

The statement $\mathcal{D} \models F$ means that formula F *holds* (i.e. is true) with respect to formula $\mathcal{D}$. That is, any model of $\mathcal{D}$ is a model of F. We call $\mathcal{D}$ the *domain* or *theory*, which decides if a given ground atomic formula is true or false.

2.2.3 Constraint Programming (CP)

Constraint Programming (CP) is a programming paradigm where a problem is expressed *declaratively* in terms of a set of *constraints*, which are *solved* to obtain a desired answer. CP is well known for its ability to find solutions to difficult combinatorial problems, e.g. timetable scheduling, relatively quickly compared with more traditional programming techniques. This is because of the interactions between unknowns through constraints, e.g. fixing an element in a timetable may fix other elements, which means the size of the search space (i.e. all potential timetables) has been significantly reduced.

There are several CP languages in existence, e.g. CHIP [23] and other Constraint Logic Programming (CLP) languages (which are covered in Section 2.2.4), OPL [42], Concurrent Constraints (CC) [69] and CHRs. There are several books on CP [61, 85, 7, 35].

Constraint programming is based on first order predicate logic, where constraints are atomic formulae. For example, the equation $X = Y + 1$ is a constraint. The predicate symbol is $=$ which represents equality, and the arguments are variable X and function call $Y + 1$. In turn, the arguments to the function call are variable Y and value 1. Another example is `leq(`A`,`B`)` from Example 1, where the predicate symbol is `leq`, and variables A and B are the arguments.

Given a set of constraints, a *constraint solver* (or simply *solver*) attempts to *solve* the constraints by applying domain specific algorithm(s). For example, the `leq` program from Example 1 solves less-than-or-equal-to constraints using the CHR execution algorithm and the rules from the program. Other examples include finite domain solvers using domain propagation, Herbrand solvers using a unification algorithm, linear arithmetic constraint solver using the simplex algorithm, and many others.

The process of solving has three possible outcomes: *satisfiable*, *unsatisfiable* or *unknown*. A set of constraints are satisfiable if there exists an assignment of values to the variables such that the constraints become true. Such an assignment is known as a *solution* to the constraints. For example, the linear arithmetic constraint $X = Y + 1$ is satisfiable because there exists at least one solution, e.g. $(X = 1 \wedge Y = 0)$. On the contrary, constraints are *unsatisfiable* if the constraints can never be made true for any assignment of values to variables. For example, the constraint $X = X + 1$ is unsatisfiable. If the constraints are unsatisfiable, then we say that *failure* has occurred. Finally, the result may be *unknown*, which means that the internal algorithms used by the constraint solver are too weak to determine if the constraints are satisfiable or not. Such solvers are *incomplete*, and are usually chosen for efficiency reasons. It is common for solvers to be incomplete, e.g. finite domain solvers are generally incomplete.

We assume that all solvers support the the trivial constraint $true$ (which is always satisfiable) and the $false$ constraint (which is always unsatisfiable).

The set of constraints handled by the constraint solver is known as the *constraint store*, which we usually represent by symbol B. A constraint solver is *incremental* if it supports the addition/removal of constraints into/from the constraint store as execution proceeds. All solvers we consider in this thesis are assumed to be incremental. Sometimes we denote the constraint store with a number i, i.e. B_i, to take into account incremental constraint solving, where B_i represents the state of the store at the i^{th} point in the execution of the program. The initial store is always $B_0 = true$, the trivial constraint. Suppose c is to be added to the store B_i, then $B_{i+1} = (c \wedge B_i)$. Sometimes we call c a *tell constraint*, because we are *telling* the solver that constraint c should hold.

The meaning of the constraint store is decided by a *constraint theory*, which we represent by symbol $\mathcal{D}$. The store B_i is satisfiable iff $\mathcal{D} \models \exists_{B_i} B_i$ holds. Likewise, it is unsatisfiable iff $\mathcal{D} \models \neg\exists_{B_i} B_i$ holds. Ideally, the constraint theory $\mathcal{D}$ should satisfy the *satisfaction completeness* property, which means for all constraints B, either $\mathcal{D} \models \exists_B B$ or $\mathcal{D} \models \neg\exists_B B$ holds.

For an incomplete solver $\mathcal{S}$, we introduce a special test $\mathcal{D} \models_{\mathcal{S}} B$, which holds if $\mathcal{S}$ can prove that $\mathcal{D} \models B$ holds (for arbitrary formula B). We consider $\mathcal{D} \models_{\mathcal{S}} B$ to have failed if either the solver can prove $\mathcal{D} \models \neg B$ holds or is too weak to prove that $\mathcal{D} \models B$ holds (i.e. we treat the *unsatisfiable* and *unknown* as the same). We (re)define (un)satisfiability for incomplete solvers by replacing $(\mathcal{D} \models)$ with $(\mathcal{D} \models_{\mathcal{S}})$ above.

Most solvers provide facilities for querying the current state of the constraint store. For example, given a solver store B, it might be useful to know if a constraint c is entailed by B. We call this an *ask constraint* (as opposed to a *tell constraint*), which holds iff $\mathcal{D} \models_{\mathcal{S}} (B \rightarrow c)$ holds. Often it is the case that a constraint solver provides procedures for both the tell and ask versions of a constraint.

2.2.4 Logic Programming (LP)

In this section we give a brief overview of Logic Programming (LP). The idea of logic programming is programming with logic, where a program is a logical statement, and the execution algorithm is a form of theorem proving. A more comprehensive introduction to LP can be found in [57].

Logic programming can be thought of as form of CP over the Herbrand domain,[1] and this is how we shall present it.

Prolog

By far the most common logic programming language in use today is Prolog. There are many implementations including [39, 88, 12, 68, 19, 15] and several

[1]The *Herbrand domain* is simply the domain of all terms.

textbooks [79, 65, 14, 10]. There is also an ISO standard [50].

The basic syntax for Prolog is as follows. Conjunction ($\wedge$) is represented by a comma '`,`' and disjunction ($\vee$) by a semicolon '`;`'. There are three types of symbols to consider: *variables*, *predicate symbols* and *function symbols*. Variables are alphanumeric string that must begin with a uppercase letter or an underscore character '`_`', and anything else is a function or predicate symbol. The value of this string distinguishes the variable (or function/predicate symbol). The exception are variables represented by a single underscore, i.e. '`_`'. Each occurrence of the underscore character in the program represents a unique unbound variable. For example, $p(\_, \_)$ is equivalent to $p(A, B)$ where A and B do not appear elsewhere in the program.

Program data in Prolog are *term*s constructed from variables and function symbols (defined the same way as terms in predicate logic). Prolog also allows integers and floating point numbers as constants.

Terms can use used to construct complex data structures, such as lists, trees, etc. Prolog uses a special notation for lists (which we borrow for our sequence notation). The empty list *nil* is represented by the atom '`[]`', and $cons(A, B)$ by the special term `[`A`|`B`]`. Apart from the special syntax, a list is an ordinary term.

Procedures in Prolog are called *predicates*. A *call* to a predicate p is represented by an atomic formula $p(t_1, ..., t_n)$ where p is a predicate symbol.

A Prolog program is a sequence of *clauses* of the form $(H$ `:-`$B_1, ..., B_n)$, where $H, B_1, ..., B_n$ are atomic formulae. The operator '`:-`' separates the *clause head* H from the *clause body* $B_1, ..., B_n$. Both the clause body and the operator '`:-`' are optional, and if missing, indicates the body is *true*. Each clause is terminated with a full stop '`.`'.

The *declarative* view of a clause $(H$ `:-`$B_1, ..., B_n)$ is the implication

$$\forall_H (H \leftarrow \bar{\exists}_H B_1 \wedge ... \wedge B_n)$$

Such an implication is called a *definite clause*, which is a special kind of *Horn clause* [49]. The declarative view reads as follows: for all values of the variables in H, if H is true, then all of $B_1 \wedge ... \wedge B_n$ must be also true for some value of the variables not in H. Thus to execute H, we must execute all of $B_1 \wedge ... \wedge B_n$. This is the *operational view* of logic programming, which is discussed below. In this thesis, we are mostly interested in the operational view of logic programming.

Prolog is also a constraint programming language with one kind of solver: the Herbrand equation solver over terms. The constraint X `=` Y is added to the constraint store by calling a special predicate (the unification predicate) of the same form, where X and Y can be any valid term. If X and Y are not unifiable, then failure occurs. Prolog also supports the built-in predicate X `==` Y, which is an ask version of the equality constraint, i.e. the call X `==` Y succeeds iff the current store entails X and Y are equal.

The operational semantics of Prolog is based on SLD-resolution (linear resolution with selection function for definite clauses), for more information see [57].

We give a simplified summary of the operational semantics as follows. The input to the program is called the *initial goal* $G_1, ..., G_m$, which is a conjunction of atomic formulae, and represents a Horn clause with an empty head, i.e. $(true \leftarrow \bar{\exists}_\emptyset G_1 \wedge ... \wedge G_m)$. Initially, the constraint store is empty, i.e. $B = true$. At each execution step we select a subgoal G_i, and a (renamed apart) clause $(H \texttt{:-} B_1, ..., B_n)$ from the program such that G_i and H are *unifiable* (i.e. the addition of the Herbrand constraint[2] $G_i = H$ into the store will not cause failure). The resulting goal is

$$B_1, ..., B_n, G_1, ..., G_{i-1}, G_{i+1}, ..., G_m$$

and the constraint $G_i = H$ is added to the store. Otherwise, if no clause with a head H unifiable with G_i can be found, then failure has occurred. In practice, Prolog always executes the subgoals from left-to-right and chooses clauses top-down (in textual order). This corresponds to a depth-first-search execution strategy.

Sometimes there may be more than one possible clause with a head unifiable with a given subgoal G_i. We call such predicates *nondeterministic*, and they are executed as follows. When subgoal G_i is selected a *choice point* is created, and the top-most eligible clause for the predicate is tried. If this choice leads to failure, then execution will *backtrack* to the choice point, and the next clause is tried, and so forth. Failure occurs once all clauses have been tried without success. Choice points can be nested, and when failure occurs, execution jumps to the "most recent" (i.e. the set of choice points forms a stack). If failure occurs and the choice point stack is empty, then the initial goal has failed.

For any given initial goal $G_1, ..., G_m$ there are three possible outcomes. The first is that failure occurs (as detailed above). The second is that the goal becomes empty, in which case we say that the goal *succeeded*, and the constraints in the Herbrand store become the *solution*. The third possibility is that the goal never becomes empty, and execution proceeds indefinitely. In this case we say that the program is *nonterminating*.

We can now give a simple example of a Prolog program.

Example 3 *The following is a simple program for the concatenation of two lists. It consists of two clauses.*

```
append([],Bs,Bs).
append([A|As],Bs,[A|Cs]) :-
    append(As,Bs,Cs).
```

Consider the initial goal `append(Xs,Ys,[1])`*. In effect we are asking for two lists* Xs *and* Ys *such that their respective concatenation results in the list '*[1]*'.*

There are two clauses that can be unified with the initial goal (i.e. `append`/3 *is nondeterministic), so a choice point is created. Prolog always chooses clauses*

[2]By treating the atomic formulae G_i and H as ordinary terms.

in textual order, so after choosing the first clause we arrive at the solution $(Xs = [] \wedge Ys = [1])$ *by unifying the goal with the clause head.*

Prolog allows the user to backtrack to find additional answers. By backtracking to the first choice point we can consider the second clause. The unification adds the following (simplified) constraint $(Xs = [1|As])$ *to the store, and calls the (new) goal* `append(`As`,`Ys`,[])`*. The new goal can only be matched with the first clause, which adds the constraint* $(As = [] \wedge Ys = [])$ *into the store. Thus the final solution is* $(Xs = [1] \wedge Ys = [])$.

Backtracking again will cause failure, since all clauses have now been tried on the initial goal. □

Prolog also supports *higher-order programming*, where goals can be stored as regular terms, and executed by a call to the special predicate `call(`*Goal*`)`. For example, the following simple program calls a given goal whenever the first argument is the number 1, otherwise failure occurs.

```
p(1,Goal) :- call(Goal).
```

The `call`/1 predicate can also be generalised to `call(`$Goal$`,`X_1`,...,`X_n`)`, where $X_1, \ldots, X_n$ are appended to the end of arguments for *Goal*, however this is not used in this thesis.

Whilst Prolog is a logic programming language, it supports several non-logical features. For example, several special *meta-predicates* are supported. A meta predicate implements some action that cannot be expressed in a finite sequence of regular clauses. For example, Prolog supports a useful meta-predicate `var(`V`)`, which causes failure if V is not a variable.

Prolog also supports more "procedural" style language constructs, such as the *if-then-else*. The syntax is `(`*Cond*`->`*Then*`;`*Else*`)`, where *Cond* is the condition, and *Then* is the *then-part* and *Else* is the *else-part*. Operationally, if the condition executes without causing failure, then the then-part is executed, otherwise the else-part is executed. Any choice points created during the execution of the condition are discarded. This has the effect of committing to the first (partial) solution generated whilst executing *Cond*.

Example 4 *For example, the following is the* `append` *program from Example 3 redefined using an* if-then-else *construct.*

```
append(As,Bs,Cs) :-
    ( As = [] ->
        Cs = Bs
    ;   As = [A|As1],
        Cs = [C|Cs1],
        append(As1,Bs,Cs1)
    ).
```

Consider the initial goal `append(`Xs`,`Ys`,`[1]`)`*. Choosing the first clause (there is only one clause to consider), we execute the condition for the if-then-else, which is*

the subgoal $Xs = []$. *This subgoal succeeds (*Xs *is unified with* $[]$*), so the then-part* $[1] = Ys$ *is executed. Thus the solution is* $(Xs = [] \wedge Ys = [1])$ *as before.*

Unlike Example 3, no choice points are created, so it is not possible to backtrack and find the second solution $(Xs = [1] \wedge Ys = [])$. *Effectively, the if-then-else commits to the first solution.* □

Mercury

The Mercury programming language [78, 77, 40] is more recent logic programming language that is very similar to Prolog. The syntax and semantics for Mercury is essentially the same as for Prolog, i.e., a Mercury program is a sequence of clauses. However, unlike most versions of Prolog, Mercury is strongly typed and strongly moded [66]. Mercury programs generally out perform equivalent Prolog programs. This is mainly because Mercury is capable of better optimisation, thanks to additional information, e.g. types and modes, available to the compiler.

Mercury's runtime term representation of terms is more restricted than Prolog. In Mercury, a term is fully a ground Prolog-style term, or a free unbound variable. For example, the Prolog term $[1, X|Y]$ would not be allowed in Mercury. This may seem to be a severe restriction, but in practice it is not. The advantage is that the Mercury compiler can use a more specialised (hence more efficient) unification algorithm than Prolog.

We delay the introduction of the Mercury type and mode system until we introduce HAL, since in HAL these are very similar.

It is worth noting that some variants of Prolog, notably the Ciao Prolog Development System [11], similarly support type and mode declarations. Like Mercury, this information is used for analysis and optimisation.

Constraint Logic Programming

Constraint Logic Programming (CLP) is the marriage of Constraint Programming (CP) and Logic Programming (LP). It is a powerful combination of constraint solving from CP with search capabilities from LP.

CLP can be thought of as a generalisation of LP, where the Herbrand solver has been replaced by constraint solvers from other domains. Usually we write CLP(S), where S is the name of the solver. Famous variants of CLP include CLP($\mathcal{R}$) [53, 89], CLP($\mathcal{Z}$) and CLP(FD) [22]. Ordinary Prolog can be thought of as CLP(H), where H is a Herbrand equation solver. More information about CLP can be found in [61, 51, 52].

2.2.5 HAL Programming Language

HAL [20, 17] is a new CLP language built on top of Mercury. In fact, operationally and syntactically, HAL is closely related to Mercury (which is related to Prolog). HAL supports more general constraint solvers, whereas Mercury has no such

inbuilt support. The current version of the HAL compiler (used in this thesis) compiles directly to Mercury.

The aims of the HAL language were to provide an efficient CLP language, with a flexible choice of constraint solvers, and various options available for defining new solvers or extending old solvers. Many solvers for HAL have been implemented. These include a a Herbrand solver [21] based on PARMA bindings [82], a bounds propagation solver (written by the author of this thesis) and a Boolean solver amongst others. Some earlier implementations of constraint solvers, e.g. CLP($\mathcal{R}$) and CLP($\mathcal{Z}$), have also been ported to HAL. For an overview of how constraint solvers are defined in HAL, see [18].[3]

For the rest of this section, we describe a brief overview of the subset of HAL needed for this thesis.

Predicates, functions and constraints

Syntactically and operationally, clauses in HAL are the same as a clauses in Prolog. In addition, HAL has notions of *functions* and *constraints*.

A call to a constraint is the same as a call to a predicate, however the former updates the constraint store. For example, in Prolog the call X `=` Y is a constraint call, which adds an equality constraint to the Herbrand solver's store. In HAL, the call X `=` Y is implemented as a predicate that updates the PARMA bindings (i.e. the representation of the constraint store) inside the HAL Herbrand solver. In general, the representation of the constraint store depends on the implementation of a solver. Typically, either some global data structure is used, e.g. a hash table of variables to constraints on the variables, or the constraints are stored within the internal representation of the variables themselves (as is the case with the Herbrand solver). Each solver defines the internal representation of the variables it uses. More information can be found in [20].

HAL also supports function calls, which makes it a strict functional language as well as a logic programming language. Function calls in HAL are treated as a special kind of predicate call with an extra argument representing the output of the function. For example, a call to the addition function `(`X `+` Y`)` is treated as the predicate call `'+'(`X`,`Y`,`Z`)`, where Z will be the result of the addition of X and Y. This translation is performed at compile time. Functions are defined using a special clause syntax, with an arrow `-->` representing function equality. For example, the following defines the function `(`X `+` Y`)` in terms of a (given) predicate `plus(`X`,`Y`,`Z`)`.

```
X + Y --> Z :- plus(X,Y,Z).
```

Functions are similarly supported in Mercury, however a different syntax is used.

[3]The paper also mentions a very early version of a CHR compiler for HAL. This version of the compiler is now obsolete.

Types

Each predicate in HAL is assigned with a *type*. The type determines what ground values arguments to the predicate are allowed to take. For example, the function `+`/2 expects either integers or floating point number arguments. Types are checked at compile time, and a error is generated if the program is inconsistent with the types declared by the programmer.

HAL provides four main *primitive types*: `int` (integer), `float` (floating point number), `char` (character) and `string` (character string). There are also *constructor types*: a *type constructor* is a functor t with some arity n, a *type expressions* is either a variable or a type constructor (of arity n) applied to $t_1, ..., t_n$ type expressions. The user defines new constructor types with the *type declaration*:

`:- typedef t(`$v_1, ..., v_n$`) -> f`$_1$`(`$t^1_1, ..., t^1_{m_1}$`) ; ... ; f`$_k$`(`$t^k_1, ..., t^k_{m_k}$`).`

where $v_1, ..., v_n$ are distinct type variables, and $t^1_1, ..., t^k_{m_k}$ are type expressions which contain at most variables $v_1, ..., v_n$. For example, the *list type* is defined as

```
:- typedef list(A) -> ([] ; [A|list(A)]).
```

HAL also allows *type renamings*, e.g.

```
:- typedef ints = list(int).
```

defines a type `ints` which is identical to a list of integers.

Types are associated to each predicate by a special '`pred`' declaration (which also declares the existence of a predicate `p`/n to the compiler).

`:- pred p(`$t_1, ..., t_n$`).`

Where `p`/n is a predicate, and $t_1, ..., t_n$ are type expressions representing the *types* of the arguments for `p`/n. Polymorphic types are allowed, i.e., $t_1, ..., t_n$ may contain variables.

In addition to the '`pred`' declaration, there are similar '`func`' and '`chrc`' declarations for declaring the types of functions and constraints[4] respectively. For example, the declarations

```
:- func float + float ---> float.
:- chrc leq(int,int).
```

give types for function `+`/2 and constraint `leq`/2 respectively. Each predicate/-function/constraint must have exactly one type declaration.

There is also a special type for higher order programming. The type `pred(`$t_1, ..., t_n$`)` matches a predicate of arity n whose arguments have types matching $t_1, ..., t_n$. For example, the type `pred(float,float,float)` will match the `plus`/3 predicate mentioned above.

HAL also supports *overloading* using *type classes* [86]. A typeclass `c`/n is defined by the following declaration.

[4]More specifically, constraints defined by a CHR solver (e.g. the `leq`/2 constraint from Example 1). The token '`chrc`' is an acronym for CHR Constraint. Constraints from other kinds of solvers simply use '`pred`' declarations.

```
:- class c(v_1, ..., v_n) where [
    <preds-and-modes>
].
```

where $v_1, ..., v_n$ are type variables, and <*preds-and-modes*> is a sequence of `pred` declarations (without the '`:-`') and `mode` declarations (see below). The predicates defined by this sequence (which are sometimes called *methods*) are *overloaded*, which means they can be defined for multiple types.

The programmer can define an *instance* to the typeclass by the following declaration

```
:- instance c(t_1, ..., t_n) where [
    pred(p_1/a_1) is q_1,
    ⋮,
    pred(p_m/a_m) is q_m
].
```

where $t_1, ..., t_n$ are non-variable type expressions, $\mathtt{p}_i/\mathtt{a}_i$ are the functor/arities of the methods (from the `class` declaration) and $\mathtt{q}_i$ is the functor of the predicate which implements the corresponding method. This means that a call to the overloaded predicate $\mathtt{p}_i/\mathtt{a}_i$ with types $t_1, ..., t_n$ will be replaced with a call to $\mathtt{g}_i/\mathtt{a}_i$ at either compile time or runtime.

Example 5 *An important typeclass in HAL is* ***`solver(T)`****, as given by the following declaration.*

```
:- class solver(T) where [
    pred init(T)
].
```

We omit the ***`mode`*** *declaration (as we have not covered them yet). The method* ***`init`****/1 creates a fresh (unbound) variable for the solver of type* ***`T`****.*

Consider a finite domain solver over the integers. The solver writer must declare an instance of the ***`solver`*** *class.*

```
:- instance solver(fdint) where [
    pred(init/1) is fd_init
].
```

Overloaded calls to ***`init(X)`*** *where* ***`X`*** *has type* ***`fdint`*** *are replaced with a call to* ***`fd_init(X)`*** *which is defined by the finite domain solver.* □

Types that have an associated instance to the `solver` class are referred to as *solver types*. For example, `fdint` is a solver type.

Modes

Each predicate/function/constraint in HAL has one or more *modes*. A simple form of modes indicate which arguments are "input" (denoted by '`in`') and which are "output" (denoted by '`out`').

Modes are declared by the programmer in the form of a `mode` declaration. For example, consider the predicate `append`/3 from Example 3. The programmer might declare the following modes.

```
:- mode append(in,in,out).
:- mode append(out,out,in).
```

The first mode indicates that given two input lists, `append`/3 produces an output list. Likewise, the second mode indicates given an input list as the third argument, `append`/3 produces output list(s) as the first and second arguments. This is the mode used in Example 3.

In general a mode indicates the before and after *instantiations* of the argument. Valid instantiations include `new`, which indicates a "new" unbound variable, and `ground`, which is a ground term. The mode `out` is defined as `(new -> ground)`, which means before the call the argument is `new`, and after the call the argument is `ground`. The mode `in` is similarly defined as `(ground -> ground)`.

Terms which contain solver variables are generally neither `new` nor `ground`, so HAL uses special instantiations (and modes) to take this into consideration. The instantiation '`old`' allows for any term, including terms that are neither a variable nor ground, e.g. $[1, X|Y]$. A variable that is `old` is also considered different than a variable that is `new`. An `old` variable has been initialised by a solver (by a call to `init`/1), whereas a `new` variable has not. For this reason, we sometimes call an `old` variable a *solver variable*.

There are two associated modes for `old`: '`oo`', which is defined as `(old -> old)`, and '`no`', which is defined as `(new -> old)`. Constraints provided by solvers use these modes, e.g. the mode

```
:- mode oo =< oo.
```

is the `mode` declaration for the constraint $X \leq Y$. Mode checking in Mercury is very similar to that in HAL, except Mercury does not support the `oo` nor `no` modes. In general, the programmer can define more complex instantiations and modes. For simplicity, we do not consider these for this thesis. More information about mode checking in HAL can be found in [37].

Determinism

Determinism may also be declared by the programmer (or inferred by the compiler). The *determinism* determines how many times a predicate may succeed on backtracking. A different determinism may be assigned to each mode of a predicate, e.g.

```
:- mode append(in,in,out) is det.
:- mode append(out,out,in) is multi.
```

The mode '`is det`' indicates that the first mode of `append`/3 is *deterministic*, i.e. succeeds exactly once and never fails. The second mode has determinism '`multi`', which means it succeeds at least once, and we may find other solutions on backtracking.

The types of determinism we are interested in for this thesis are '`det`' (succeeds once), '`semidet`' (succeeds at most once) and '`failure`' (never succeeds). Constraints generally have `semidet` determinism, with the exceptions being the *true* constraint (which is `det`) and the $false$ constraint (which is `failure`). For predicates, other types of determinism are supported. These are '`multi`' (succeeds at least once) and '`nondet`' (succeeds any number of times).

Determinism inference/checking is exactly the same in Mercury as it is in HAL. More information about determinism analysis in Mercury (and hence in HAL) can be found in [41].

Example program

The following is the HAL version of `append` program from Example 3.

```
:- pred append(list(A),list(A),list(A)).
:- mode append(oo,oo,oo) is nondet.
append([],Bs,Bs).
append([A|As],Bs,[A|Cs]) :-
    append(As,Bs,Cs).
```

The definition of `append`/3 itself is identical to that in Example 3. The `pred` declaration means that all three arguments to the `append` predicate must be of type `list(`A`)` for the same A. This means that the type of the elements of all three lists must be the same. The modes for each argument is `oo`, which means that the arguments can be any term that matches the type, and may contain solver variables. The determinism is `nondet`, since depending on the input, zero or more solutions are possible.

2.2.6 Future of HAL

The version of HAL that is used throughout this thesis has been discontinued. Instead, the Mercury programming language is in the process of being adapted to implement the core ideas of HAL directly (i.e. native support for constraint solvers). There are several benefits for merging the two languages, since type, mode and determinism analysis happens once per compilation, as opposed to twice under the old system (once by the HAL compiler, then again by the Mercury compiler). There are no major consequences for this thesis, although there may be differences in syntax. In future, a CHR compiler for Mercury will be written, using and extending the ideas in this thesis.

2.3 Constraint Handling Rules

CHRs are a high-level CP language for specifying and extending constraint solvers. Before CHRs, constraint solvers were implemented in low-level languages, such as `C` or `C++`. CHRs offer a far more declarative approach, where the program closely corresponds to the declarative semantics of the intended solver. For example, the `leq` solver from Example 1 is expressed very concisely in CHRs.

In this section we give an overview of CHRs. We will cover both the theoretical and practical aspects of the CHR programming language, related systems and the history of CHR implementations.

2.3.1 Syntax and Semantics

CHRs are a CP language which use existing *built-in constraints* to define new *CHR constraints*. The built-in constraints are provided by one or more *built-in* solvers. For example, the `leq` program from Example 1 uses a built-in Herbrand solver, and the equations X `=` Y are the built-in constraints. In general, any constraint solver (e.g. finite domain, Boolean, etc.) may be used as a built-in solver. The constraints defined by the CHR rules are the CHR constraints. For example, the constraint `leq(`X`,`Y`)` is a CHR constraint.

CHR constraints are defined by three types of rules: *simplification*, *propagation* and *simpagation* rules, and the respective syntax is

$$\begin{array}{lll} \mathit{name} \texttt{ @ } h_1, \ldots, h_n & \texttt{<=>} & g \texttt{ | } b_1, \ldots, b_m \texttt{.} \\ \mathit{name} \texttt{ @ } h_1, \ldots, h_n & \texttt{==>} & g \texttt{ | } b_1, \ldots, b_m \texttt{.} \\ \mathit{name} \texttt{ @ } h_1, \ldots, h_l \texttt{ \textbackslash } h_{l+1}, \ldots, h_n & \texttt{<=>} & g \texttt{ | } b_1, \ldots, b_m \texttt{.} \end{array}$$

where *name* is the rule name, $h_1, \ldots, h_n$ are CHR constraints, g is a conjunction of built-in constraints, and $b_1, \ldots, b_m$ is a conjunction of CHR constraints and built-in constraints. Both *name* and g are optional. Simplification and propagation rules are differentiated by the type of arrow: '`<=>`' for simplification, and '`==>`' for propagation. Simplification and simpagation are differentiated by a backslash '\' token in the rule head. Like Prolog clauses, CHR rules are always terminated with a full stop '`.`'. A *CHR program* P is a finite sequence of CHR rules.

The CHR constraints $h_1, \ldots, h_n$ are the *head* of a rule. Sometimes we may also refer to each individual h_i as a *head*. The conjunction of built-in constraints g is the *guard* of a rule, and $b_1, \ldots, b_m$ are the *body* of the rule. The head and body of a rule must be non-empty. If the guard g is omitted, then this is equivalent to $g \equiv true$.

We say that a rule r *fires* if we apply r to some subset of the CHR constraints in the store. We use the notation $(S \rightarrowtail S')$ to represent a rule firing on the constraint store S to give the new constraint store S'. The conditions for a rule firing are

1. there exists a multiset of constraints $\{h'_1, ..., h'_n\} \subseteq S$ that *matches* the head $h_1, ..., h_n$; and

2. the guard g is entailed by S.

Matching is very similar to unification, except the variables in $h'_1, ..., h'_n$ are only allowed to be bound to variables in $h_1, ..., h_n$. Matching can be thought of as one way unification. For example, constraint `leq(`A`,`B`)` does not match the head `leq(`X`,`X`)`, since the equations $(A = X \wedge B = X)$ bind A and B to the same value. Note that `leq(`X`,`X`)` does match `leq(`A`,`B`)`, since the equations $(X = A \wedge X = B)$ only bind X to variables in `leq(`A`,`B`)`. The set of equality constraints generated from the matching is the *matching substitution* which is usually represented by θ. A matching substitution θ can be *applied* to a term T, which means that variables aliased by θ that do not appear in S are eliminated.

The action performed when a rule fires depends on the type of rule.

- *Simplification*: Given a constraint multiset $\{h'_1, \ldots, h'_n\}$ and a matching substitution θ such that $\{h'_1, \ldots, h'_n\} = \theta(\{h_1, \ldots, h_n\})$, and $\theta(g)$ is entailed by S, then we *replace* $\{h'_1, \ldots, h'_n\}$ with $\theta(\{b_1, \ldots, b_m\})$.

- *Propagation*: Given a constraint multiset $\{h'_1, \ldots, h'_n\}$ where $\theta(g)$ is entailed by S, we *add* $\theta(\{b_1, \ldots, b_m\})$.

- *Simpagation*: Given a constraint multiset $\{h'_1, \ldots, h'_n\}$ where $\theta(g)$ is entailed by S, we replace $\{h'_{l+1}, \ldots, h'_n\}$ (the constraints matched *after* the '\' token) by $\theta(\{b_1, \ldots, b_m\})$.

A sequence of rule applications is called a *derivation*.

The operational semantics of CHRs is to exhaustively fire rules on an initial goal until no further rule is application is possible.

Example 6 *The following is a similar derivation to that of Example 2, except the initial goal has two copies of the constraint $B \leq A$. The type of rule application for each step has been marked.*

$$\begin{array}{ll} & A \leq B, C \leq A, B \leq C, B \leq C \\ \rightarrowtail_{simpagation} & A \leq B, C \leq A, B \leq C \\ \rightarrowtail_{propagation} & A \leq B, C \leq A, B \leq C, C \leq B \\ \rightarrowtail_{simplification} & A \leq B, C \leq A, B = C \\ \rightarrowtail_{simplification} & A = B, B = C \end{array}$$

The simpagation corresponds to the `idempotence` *rule, propagation with the* `transitivity` *and simplification with the* `antisymmetry` *rule.* □

Often, several rules can fire on any given store, in which case a rule is chosen *nondeterministically*. This inherent nondeterminism makes *confluence* important, which we will discuss in the next section.

A considerably more formal definition of the operational semantics of CHRs will be covered in Chapter 3. Alternatively, see [32, 1]. CHRs also have a declarative semantics, which will also be discussed in Chapter 3.

In past literature, e.g. [32, 1], simpagation rules are treated as simplification rules, i.e.,

$$h_1, \ldots, h_l, h_{l+1}, \ldots, h_n \texttt{ <=> } g \texttt{ | } h_1, \ldots, h_l, b_1, \ldots, b_m\,.$$

An alternative approach is to treat simplification and propagation rules as special cases of simpagation rules, where no constraint appears before or after the backslash '\'. Let ϵ represent the empty sequence, then

$$h_1, \ldots, h_l \setminus \epsilon \texttt{ <=> } g \texttt{ | } b_1, \ldots, b_m\,.$$
$$\epsilon \setminus h_1, \ldots, h_l \texttt{ <=> } g \texttt{ | } b_1, \ldots, b_m\,.$$

are a propagation and simplification rule respectively.

The operational semantics prevent a propagation rule firing more than once on a given set of constraints. The purpose of this restriction is to prevent trivial nontermination by repeatedly applying the same rule. For more information see [1], or Chapter 3.

2.3.2 Confluence and Termination

CHRs are a *committed choice* language. This means that, unlike Prolog, no choice points are created and there is no backtracking (to try other rules) on failure. In effect, we are committing to a choice of rule, even if several possibilities exists. This is also called *don't care nondeterminism* (committed choice) as opposed to *don't know nondeterminism* (search + backtracking).

As with all systems that use *don't care nondeterminism*, the property of *confluence* is essential. Confluence is informally defined as follows: For all possible stores S, if we can apply a rule to get S', and apply a (possibly different) rule to get S'', then the resulting final stores after executing S' and S'' must be *variants*. Two stores are variants if they are identical up to the renaming of new variables introduced during the execution of S' and S''. Precise definitions of confluence and related topics will be covered in Chapter 3. Even more informally, *confluence* is the property that the program always computes the same output for every given input (no matter which derivation is chosen).

Example 7 *The* `leq` *program in Example 1 is an example of a confluent program. The following is an example of a non-confluent program.*

```
p <=> q.
p <=> r.
```

Starting from the initial goal of a single `p` *constraint, the final store will be either* `q` *or* `r` *depending on which rule is chosen.* □

Confluence for CHRs has been extensively studied. In [32, 4, 1] a decidable confluence test for terminating CHR programs is presented. The property of confluence has been recognised before in other rule based languages. For example,

confluence has been well studied in relation to term rewriting systems [9]. Other systems rely on a weaker form of confluence, known as *determinism* (e.g. see [59] for definition). Determinism is a trivial form of confluence, i.e. when only one rule is applicable to a given store.

Another important property for CHR programs is *termination*. The general problem of termination of CHR programs has been studied in [33]. Termination is an undecidable property, however several common CHR programs have been shown to be terminating. Some common CHR solvers are nonterminating.

Example 8 *Surprisingly, the famous* `leq` *program from Example 1 is nonterminating. Consider the following initial goal.*

$$A \leq A, A \leq A$$

Since CHRs use a multiset semantics, multiple copies of the same constraint (in this case $A \leq A$) are allowed. We achieve an infinite derivation by continually applying the `transitivity` *propagation rule to the first constraint, and the rightmost constraint.*[5]

$$\begin{array}{ll} & A \leq A, A \leq A \\ \rightarrowtail_{transitivity} & A \leq A, A \leq A, A \leq A \\ \rightarrowtail_{transitivity} & A \leq A, A \leq A, A \leq A, A \leq A \\ \ldots & \end{array}$$

Even with a fair rule application, it is possible to construct a nonterminating derivation for the `leq` *program.* □

The question of confluence for terminating CHR programs is decidable, as shown by the following theorem.

Theorem 1 *[32, 1] A terminating CHR program is confluent iff all its critical pairs are joinable.*

In this theorem, a *critical pair* is a pair of constraint stores constructed as follows. Given two (not necessarily different) CHR rules r_1 and r_2 we find a most general (and minimal) store S such that S can fire both r_1 and r_2. Such an S is called a *non-trivial direct common ancestor state* of r_1 and r_2.

Example 9 *Consider the following two rules from the* `leq` *programs.*

```
leq(X,Y), leq(Y,X) <=> X = Y.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Then the following store S

```
leq(A,B), leq(B,A)
```

[5]Note that multiple copies of constraints are considered different in CHRs.

is a non-trivial direct common ancestor state. Note that both rules can be applied to S.

There are typically several such non-trivial direct common ancestor states. For example

```
leq(A,B), leq(B,A), leq(B,C)
```

is another such state. □

More formally, a non-trivial direct common ancestor state is constructed by partitioning the heads $H_1 = H'_1 \uplus H''_1$ from r_1 and $H_2 = H'_2 \uplus H''_2$ from r_2, then we construct $S = H'_1 \wedge H''_1 \wedge H''_2 \wedge (H'_1 = H'_2)$ provided H'_1 and H'_2 are unifiable. See [32] for more details.

Let S_1 and S_2 be the results of applying r_1 and r_2 to a non-trivial direct common ancestor state S respectively, then (S_1, S_2) is a *critical pair*. The critical pair is *joinable* if executing S_1 and S_2 results in variant final states. If all critical pairs are joinable, then by Theorem 1 the program is confluent. We note that there are finitely many critical pairs for any given program, hence testing for confluence for terminating programs is decidable.

If a CHR program is not confluent, then it is sometimes possible to automatically derive a (declaratively) equivalent confluent program. This process is known as *completion*. Completion for CHRs has been studied in [3].

In Chapter 3 we give considerably more formal definitions of confluence, termination, critical pairs and other related concepts.

2.3.3 History of Implementations

In this section we give an overview of all (known) CHR implementations to date, roughly in chronological order.

Original implementations

CHRs were invented (and first implemented) in 1991 by Thom Frühwirth [30, 31]. Originally, CHRs were known as *Constraint Simplification Rules*, or SiRs for short. The syntax and operational semantics of SiRs are nearly identical to the modern day CHRs, however there are some differences. For example, SiRs did not have *simpagation* rules.[6] Also, SiRs used (now redundant) '`callable`' declarations whose purpose was to declare the conditions a CHR constraint can be "called", i.e. considered for matching against rules. An example `callable` declaration from [31] is

```
callable max(X,Y,Z) if bound(X), bound(Y).
```

[6] SiRs did have both simplification and propagation rules. Simplification rules were named *replacement simplification rules*, and propagation rules were named *augmentation simplification rules*.

which states that the constraint `max`(X, Y, Z) must only be "called" if both X and Y have been *bound*, i.e. are not variables. Modern CHR implementations do not use `callable` declarations, and CHR constraints are always called eagerly.

The first actual implementation of CHRs/SiRs was a simple interpreter written for ECL^{i}PS^e Prolog in 1991. Unfortunately, this original CHR implementation is believed to have been lost for all time. However parts of the implementation, including a brief description, can be found in the appendix of [36].

The original implementation grouped rules into two classes: *single-headed* and *multi-headed* rules. Single-headed rules had a single CHR constraint in the head (e.g. the `reflexivity` rule from Example 1), and multi-headed had two constraints. Rules with more than two heads were not allowed. Some later CHR implementations have similar restrictions. Rules with large heads can be handled (by the programmer) by introducing auxiliary constraints that perform a partial match. For example, the rule

```
p(X), q(Y), r(Z) <=> true
```

can be implemented as

```
p(X), q(Y)    ==> pq(X,Y).
pq(X,Y), r(X) <=> true.
```

There are obvious problems with this approach, since the two programs are technically different (i.e. the answer to a goal may have redundant `pq`/2 constraints). Modern CHR compilers allow rules with arbitrarily large heads.

The execution algorithm of the original implementation is very different to that of modern CHR compilers. Firstly, all single-headed rules are considered before multi-headed rules. Secondly, testing the guard for single-headed rules has three possible outcomes: `success` if the guard succeeds, `failure` if the (logical) negation of the guard succeeds and `suspend` if there is not enough information to decide if the guard succeeds or fails. The third case occurs when the variables in the guard are not bound enough to decide the outcome.

Example 10 *Consider a rule with the guard $X < Y$. Given a matching substitution $(X = 3 \wedge Y = 5)$ we have enough information to decide that this guard succeeds, thus* ***`success`*** *is returned. Similarly, a matching substitution $(X = 5 \wedge Y = 3)$ causes the guard to return* ***`failure`****.*

Now consider the matching substitution $(X = 3)$. The guard neither succeeds nor fails because this depends on the value of Y. Hence the result is ***`suspend`****, which means the rule should be reconsidered when Y becomes more bound.* □

If the guard suspends, the rule is (re)scheduled (into a queue of rules) for consideration later. Modern implementations treat `failure` the same as `suspend` for simplicity.

Another very early implementation of CHRs was written for the LISP programming language.[7] It was also a simple implementation that lacked simpaga-

[7]If the reader understands German, see [43], otherwise see [32] (a secondary source) for a brief overview.

tion rules.

Next generation implementation

A significantly more sophisticated implementation of CHRs was built for ECL^{i}PS^e Prolog in 1994 [36]. This implementation is the first CHR compiler (as opposed to an interpreter), and is the ancestor of modern CHR compilers. Several improvements were made: such as Prolog calls being allowing in the rule body and *deep* guards. A deep guard contains CHR constraints (which is normally disallowed).

The ECL^{i}PS^e implementation compiles CHRs in several phases. The first phase is to translate a CHR program into an equivalent program consisting of only single-headed simplification rules with deep guards. For example, the rule

```
p(X), q(X) <=> ground(X) | true.
```

is translated to

```
p(X,H1,Id1) <=> delayed_constraint(q(X,H2,Id2)),
   ground(X), var(Id1), var(Id2) |
        remove(q(X,H2,Id2)).
```

A similar single-headed simplification rule is created with `p` replaced by `q` and vice versa. Here `delayed_constraint`/1 unifies the argument with a matching constraint from the store. If there is more than one potential match, then it returns others on backtracking. The `remove`/1 call explicitly removes a constraint from the store. Notice that two additional arguments `H` and `Id` has been added to each constraint. The argument `Id` is call the *constraint identifier*, and is an essential part of modern CHR systems. The constraint identifier is either an unbound variable or a ground atom. An atom means the constraint has been deleted, otherwise the constraint has not been deleted. The `var`/1 tests in the guard make sure the rule does not fire on deleted constraints. The additional arguments `H1` or `H2` are *propagation histories*, which are described below.

The translation for propagation rules is similar. The rule

```
p(X), q(X) ==> ground(X) | true.
```

is translated to

```
p(X,H1,Id1) <=> delayed_constraint(q(X,H2,Id2)),
   ground(X), var(Id1), var(Id2),
   not_member(n-q(X)-2,H1), not_member(n-p(X)-1,H2) |
        p(X,[n-q(X)-2|H1],_).
```

As before, a similar rule is also generated with `p` replaced by `q` and vice versa. The propagation histories for each constraint must be checked by the guard before the rule fires. This prevents a propagation rule from firing twice on the same set of matching constraints, which is disallowed by the operational semantics of CHRs.

In this implementation, a propagation history is simply a list of entries of the form n-C-p, where n is the name of the rule, C is the matching partner constraint, and $p \in \{1, 2\}$ indicates whether C matches the first or second head. The auxiliary predicate `not_member(`E`,`Ls`)` fails if entry E is present in list Ls. Note that when the rule fires, we only need to update the history for `p`. This is safe because both the histories for `p` and `q` are considered by the guard.

A second compilation phase translates the single-headed simplification rules into Prolog clauses. This is reasonably straightforward since the single-headed rule ($H \iff G \mid B$) roughly corresponds to the clause

```
H :-
    ( G ->
        remove(H), B
    ;   true
    ).
```

This translation is simplified, and in practice several subtleties must be taken into consideration. For example, executing the guard G must not bind any variables from the constraints that matched the head of the rule. This is achieved by using a special predicate `delay(`Ls`,`$Goal$`)`, which is similar to `call(`$Goal$`)`, except $Goal$ is immediately called once any variable appearing in the list Ls has *changed*. We informally define *changed* as meaning "further constrained".[8] To ensure that variables are not bound by the guard, we call `delay(`Hs`,fail)`, where Hs are all variables from the matching constraints. Thus if one of Hs changes, then `fail` (the $false$ constraint) is called. After the guard has been tested, the delayed `fail` goals are removed.

The `delay`/2 predicate is also used for *waking up* CHR constraints once a variable it contains changes. A woken up constraint rechecks every rule, since a change in a variable may result in a guard succeeding where it previously had "failed" (i.e. returned `suspend` under the original implementation, see Example 10). Consider the `leq`/2 constraint from Example 1, the following code is generated.

```
leq(X,Y) :-
    extract_vars(leq(X,Y),Vars), delay(Vars,leq(X,Y)), ...
```

This ensures that a `leq`/2 constraint is called (i.e. woken up) each time a variable in the constraint changes. Modern CHR compilers use similar techniques.

Operationally, the ECL^{i}PS^e implementation is different than that of the earlier CHR interpreters. One important difference is the notion of an *active constraint*, which represents a "call" to a CHR constraint. When a rule fires, the (current) active constraint must always be included in the set of matching constraints. Active constraints form a stack, where a call (e.g. in a rule body) pushes an active constraint onto the stack, and it is popped off once all rules have been

[8] Much later in this thesis, Definition 48 gives a formal definition of "changed".

tried. The notion of an active constraint was not deliberately invented, but it emerged as result to the way CHRs are compiled, i.e. CHR constraints to Prolog clauses. The active constraint stack is merely the program stack for Prolog. All modern CHR compilers use a similar call-based active constraint implementation.

Modern Prolog CHR compilers

There are *modern* CHR compilers for SICStus, Yap, hProlog, XSB and SWI Prolog. The SICStus and Yap Prolog versions are based on a compiler written in 1998 by Christian Holzbaur [46, 47], which represents the first modern CHR compiler to be built. The hProlog CHR compiler is based on a CHR compiler written in 2002 by Tom Schrijvers [71] called the "K.U.Leuven CHR system", which was later ported to XSB [74] and SWI Prolog [75]. Both implementations are similar, except the Schrijvers compiler uses several kinds of program analysis and optimisation (some of which are detailed in this thesis) to improve the runtime performance of the CHR programs.

Both of these compilers are considered *modern* for two main reasons. The first is because they represent the state-of-the-art in terms of CHR implementations (although this is more true for the Schrijvers compiler). The other reason is because of a more guaranteed operational behaviour of CHR programs. As a result, CHRs can be used more as a generic committed choice rule-based language, rather than a specialised language for implementing constraint solvers.

Modern CHR compilers offer the following guarantees about the operational behaviour of CHR programs:

1. A call-based *active constraint* semantics for CHRs, similar to that of the earlier ECL^{i}PS^e compiler;

2. The active constraint always checks rules in *textual* order, as specified by the programmer;

3. The bodies of rules are also executed in *textual* order, i.e. from left to right.

This means the the operational behaviour of CHRs is analogous to that of Prolog, i.e. top-down and left-to-right. In addition, some of the restrictions that existed in earlier implementations have been abolished. Notably, rules can now have arbitrary sized heads.

The new guaranteed operational behaviour of CHRs is, in effect, defining a new (more deterministic) operational semantics for CHRs. This has proved to be useful for CHR programmers, who can better visualise the control flow in CHR programs.

Example 11 *For example, consider the following simple CHR program, which writes a message to the screen.*

```
p(_) ==> write("hello ").
p(X) <=> write(X).
```

Consider the goal `p("world")`. *This program is not confluent, since executing the goal could result in either "*`hello world`*" or "*`world`*" been printed, depending on which rule is tried first. However, under the new CHR operational semantics, execution proceeds as follows. First, the constraint* `p("world")` *becomes the* active constraint *that we are currently considering for execution. Rules are tried in* textual *order, hence the first rule will always be chosen. The first rule fires, and prints "*`hello `*". Since the propagation rule has fired (and it is not allowed to fire again because of the propagation history), the active constraint proceeds to the next rule. This rule fires, and prints "*`world`*" onto the screen.*

In effect, this program is confluent under the new operational semantics, since there is only ever one output for any given input. □

This example also shows how Prolog calls, e.g. `write(X)`, can be embedded in CHR rules.

The compiler and runtime system of the modern Prolog compilers has also improved over earlier attempts. The biggest improvement is the use of *attributed variables* [44]. The functionality of attributed variables subsumes that of `delay`/2 predicate used in the early ECL^{i}PS^e compiler, i.e. it provides a means for constraints to be woken up when a set of variables has changed.

An attributed variable is equivalent to an ordinary Prolog variable with with one or more *attributes* attached to it. An attribute is any ordinary Prolog term $f(X_1, ..., X_n)$ where the functor/arity f/n uniquely defines the attribute. For example, in the SICStus Prolog attributed variable interface [39],[9] attributes are associated to a variable V by the special predicate `put_atts(`V`,`*Attribute*`)`. There is also a similar predicate `get_atts`/2 for retrieving attributes. Some constraint propagation solvers, e.g. the finite domain solver in [39], are implemented directly using attributed variables (the variable's domain is stored as an attribute).

The Prolog CHR compilers utilise attributed variables by suspending CHR constraints in an appropriate attribute attached to all variables appearing in that constraint. For example, the constraint `leq(`X`,`Y`)` is suspended inside an attribute `leqs(`Ls`)`, where Ls is a list of `leq`/2 constraints, for variables X and Y. Note that this is a slight simplification, as the attribute would also carry other information, see [46, 47] for details. The task of searching for matching partners is made more efficient by using attributed variables. For example, consider the rule

```
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

and consider the left-most occurrence of `leq`/2 constraint in the head of this rule. Given the active constraint `leq(`A`,`B`)` we require all matching partners of the form `leq(`B`,_)`. Such partners must be inside the attribute `leqs(`Ls`)` of variable B, which is quickly obtained by calling `get_atts`/2. Since Ls is usually significantly smaller than entire constraint store, the search for matching partners is very efficient. This technique is known as *attributed variable indexing*.

[9] Note that other Prologs use a different interface for attributed variables.

Attributed variable indexing does not work on ground constraints, since there are no variables to attach attributes to. In such cases, the Prolog implementations use some form of secondary indexing in order to find matching partners. This is typically either a list or (in the newer compilers) hash tables.[10]

For the rest of this section, we shall briefly describe some of the other CHR implementations currently in existence.

Bootstrapping CHR compiler

In 2002, Christian Holzbaur implemented a *bootstrapping* CHR compiler, i.e. a CHR compiler implemented in CHRs, for SICStus Prolog. Operationally, it is equivalent to the current (non-bootstrapping) compiler. This version of the compiler was never released.

JACK

JACK (JAva Constraint Kit) [5, 55] is an implementation of CHRs for the Java programming language [54]. The JACK package consists of a language to write CHRs (JCHR), a CHR visualisation tool (VisualCHR), and a generic search engine (JASE). Special declarations are used by the programmer to specify rules and goals.

The implementation of JACK is very similar to the Prolog implementations of CHRs. The main difference is that much of the functionality of Prolog had to be implemented in Java, for example, Prolog style variables, etc. The operational semantics of JCHR are different from that of the Prolog implementations.

Chameleon

Chameleon is a functional programming language [81] very similar to Haskell [56]. The main difference is that Chameleon uses an experimental implementation of type classes using a CHR interpreter. See [80] for the theory behind this approach. The idea is that type classes are represented as CHR constraints, and class and instance declarations as CHR rules. For example, the Haskell class declarations (with `ghc` extensions [83])

```
class P x y
class Q x | -> x
instance (Q x, Q y) => P x y
```

correspond to the CHR program

```
p(X,Y) <=> q(X), q(Y).
q(X), q(Y) ==> X = Y.
```

It is also possible to write CHR rules directly in a Chameleon program using a special '`rule`' declaration, e.g.

[10] Using hash tables for indexing in covered in Chapter 7.

```
rule P x y <=> Q x, Q y
rule Q x, Q y ==> x = y
```

These declarations are equivalent to the class declarations above.

The original version of Chameleon used a different operational semantics than other CHR implementations. Basically, the interpreter would iterate through all rules, exhaustively applying the current rule before moving to the next rule. It would repeat this process until the constraint store survives an entire iteration of rules without any firing. Recently, the CHR engine in Chameleon has been replaced with a version which has an operational semantics very close to that of the Prolog implementations.[11]

HAL

The first implementation of CHRs for HAL was written by Christian Holzbaur in 2001. This version was featured in [48]. It is essentially a port of the (non-bootstrapping) SICStus CHR compiler. The compiler could not handle CHR constraints with solver variables.

A newer CHR compiler for HAL was written in 2002 by the author of this thesis, and incorporates the ideas of the thesis to improve CHR compilation. Operationally, this version of the compiler is the same as that of the modern Prolog CHR compilers.

2.3.4 Related Systems

CHRs are part of the *rule based* programming languages paradigm. In all such languages, a program is a set or sequence of rules. Executing the program involves continually applying rules to an initial goal until a final state is reached. Unlike CHRs, most rule based languages are not CP languages, so there is no concept of a built-in solver.

In this section we briefly cover some of the more common/relevant rule-based languages.

Reduction systems and term rewriting

A *reduction system* is a set of *rewrite rules* over some formal domain, e.g. strings, graphs, etc. A common kind of a reduction system is a *term rewriting system* (TRS), where the rewrite rules apply to terms. See [9] for a more detailed introduction to TRSs.

In reduction systems a *rewrite rule* is of the form $(H \rightarrow B)$, which is analogous to the CHR rule $(H \iff B)$ (single headed simplification rule with no guard). Note that the rewrite rule $(H \rightarrow B)$ can be thought of as defining equality between terms, i.e. $(H = B)$.

[11] This engine was implemented by the author of this thesis.

Example 12 *A classic example of a TRS program is addition using successor notation, i.e. the integer* 1 *is represented as* $s(0)$*,* 2 *as* $s(s(0))$*, and so on.*

$$\begin{array}{ll} 0 + X & \rightarrow X \\ s(X) + Y & \rightarrow s(X + Y) \end{array}$$

The initial goal $(s(0) + 0 + s(s(0)))$ *(which represents* $(1 + 0 + 2)$*), will be reduced to* $s(s(s(0)))$ *(which represents* 3*).* □

Several implementations of TRSs exist, such as OBJ3 [38] and Maude [13] amongst others.

There are several differences between CHRs and TRSs. Unlike CHRs, standard term rewriting does not have any notion of propagation or simpagation rules, and all rules are effectively single-headed and lack guards. The goal is always a single term (rather than a conjunction of terms/constraints). Another important difference is that term rewriting rules may be applied to any sub-term of the goal, whereas CHRs do not operate on arguments to constraints. For example, in Example 12 it is possible to rewrite the goal $s(0) + 0 + s(s(0))$ to $s(0) + s(s(0))$ by applying the first rule to the subterm $0 + s(s(0))$. On the contrary, CHRs distinguish between constraints, and the arguments of constraints (CHR rules cannot be applied to the latter).

Confluence is also an important issue in term rewriting (and other reduction systems) for the same reason it is important for CHRs. In fact, the confluence results for CHRs are adapted versions of similar results for term rewriting. For example, proving confluence for term rewriting involves showing that all critical pairs are joinable. The completion algorithm for CHRs [3] is also based on a similar algorithm for term rewriting.

Guarded rules

Guarded rules [76] are highly related to CHRs. A guarded rule is of the form

$$g \mathbin{\square} h \mathbin{\triangleright} B$$

where g is the guard, H is the head (a single constraint) and B is the body (a conjunction of constraints).

Example 13 *The following is a very simple program defining a (one way)* `not`*/2 constraint for a simple Boolean solver.*

```
X = 0 □ not(X,Y) ▷ Y = 1
X = 1 □ not(X,Y) ▷ Y = 0
```

The constraint `not(1,Y)` *will be reduced to* $Y = 0$*.* □

In fact, guarded rules are equivalent to single-headed simplification rules ($h \iff g \mid B$). Thus, CHRs are more general.

Production systems and RETE/TREAT

Production systems, such as OPS5 [28], are a rule based language similar to CHRs, and are mainly used in artificial intelligence (AI) applications. In a production system a *token* is analogous to a CHR constraint, and the *working memory* is analogous to the CHR store. A *production* is similar to a propagation rule. Tokens are deleted explicitly, rather than implicitly by matching a certain type of rule (as is the case with CHRs).

RETE is an algorithm for compiling rules from a production system into a special type of optimised decision graph (sometimes called the *RETE network*). There is two types of input to the graph, $+G$ for adding the token G to the working memory, and $-G$ for explicitly removing it. A RETE network has three types of nodes: *constant test nodes* for testing if G is in a certain form, *two input nodes* for testing if a condition holds between two tokens, and *memory nodes* for storing partial matches. Adding $+G$ causes G to descend the RETE network as far as possible. Memory nodes are updated with partial matches including G. Removing $-G$ is similar, except partial matches are removed from memory nodes. For more information see [29].

The TREAT algorithm [63] is an alternative to RETE. Unlike RETE, TREAT does not store partial matches, and matchings are calculated from scratch each time $+G$ is added. The advantages of TREAT over RETE are better memory usage (no partial matches are stored) and faster deletion. In practice, TREAT generally outperforms RETE.

The matching algorithm used in modern CHR compilers is essentially equivalent to TREAT.[12] Some of the issues related to TREAT, such as the order joins are computed, are also issues for CHRs (and will be discussed later in Chapter 7).

Adapting RETE for CHRs is still an area of potential future research. The biggest problem is adapting the algorithm such that it can handle CHR-style guards which can "change" (e.g. initially fail, but then succeed when variables become more bound).

[12]Note that the TREAT algorithm handles other types of matchings not relevant to CHRs, e.g. matching rules with the condition that a token is *not in* working memory.

Chapter 3

Operational Semantics

3.1 Introduction

The operational semantics of CHRs are to exhaustively apply a set of rules to an initial set of constraints until a fixed point is reached. We refer to these operational semantics as *theoretical*, because they describe how CHRs are allowed to behave in theory. The theoretical semantics is highly nondeterministic, and this is sometimes inconvenient from a programming language point of view. For example, the theoretical semantics do not specify which rule to apply if more than one possibility exists, and the final answer may depend on such a choice.

This chapter defines the *refined* operational semantics for CHRs: a more specific operational semantics which has been implicitly described in [46, 47], and is used by almost all modern implementations of CHRs we know of (see Section 2.3.3). Some choices are still left open in the refined operational semantics, however both the order in which constraints are executed and the order which rules are applied, is decided. Unsurprisingly, the decisions follow Prolog style and maximise efficiency of execution. Later, Chapter 6 discusses the consequences of the remaining choices left open in the refined semantics. Most of the content in the rest of this thesis, especially CHR program analysis and optimisation, assumes the refined operational semantics, hence the need for a formal definition.

It is clear that CHR programmers take the refined operational semantics into account when programming. For example, some of the standard CHR examples are non-terminating under the theoretical operational semantics.

Example 14 *Consider the following simple program that calculates the greatest common divisor (*gcd*) between two integers using Euclid's algorithm:*

```
gcd1 @ gcd(0) <=> true.
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

Rule `gcd1` *is a simplification rule. It states that a fact* `gcd(0)` *in the store can be replaced by true. Rule* `gcd2` *is a simpagation rule, it states that if there are two facts in the store* `gcd(`n`)` *and* `gcd(`m`)` *where* $m \geq n$*, we can replace the part*

after the slash `gcd(`m`)` *by the right hand side* `gcd(`$m - n$`)`.[1] *The idea of this program is to reduce an initial store of* `gcd(A), gcd(B)` *to a single constraint* `gcd(C)` *where* `C` *will be the* gcd *of* `A` *and* `B`.

This program, which appears on the CHR webpage [70], is nonterminating under the theoretical operational semantics. Consider the constraint store `gcd(3), gcd(0)`. *If the first rule fires, we are left with* `gcd(3)` *and the program terminates. If, instead, the second rule fires (which is perfectly possible in the theoretical semantics),* `gcd(3)` *will be replaced with* `gcd(3-0) = gcd(3)`, *thus essentially leaving the constraint store unchanged. If the second rule is applied indefinitely (assuming unfair rule application), we obtain an infinite loop.* □

In the above example, trivial non-termination can be avoided by using a *fair* rule application (i.e. one in which every rule that could fire, eventually does). Indeed, the theoretical operational semantics given in [32] explicitly states that rule application should be fair. Interestingly, although the refined operational semantics is not fair (it uses rule ordering to determine rule application), its unfairness ensures termination in the `gcd` example above. Of course, it could also have worked against it, since swapping the order of the rules would lead to nontermination.

The `leq` program of Example 1 is another example that relies on the refined operational semantics to terminate, as was shown in Example 8. In this case, the program is nonterminating even with a fair rule application.

The refined operational semantics allows us to use more programming idioms, since we can now treat the constraint store as a queryable data structure.

Example 15 *Consider a CHR implementation of a simple database:*

```
l1 @ entry(Key,Val) \ lookup(Key,ValOut) <=> ValOut = Val.
l2 @ lookup(_,_) <=> fail.
```

where the constraint `lookup` *represents the basic database operations of key lookup, and* `entry` *represents a piece of data currently in the database (an entry in the database). Rule* `l1` *looks for the matching entry to a lookup query and returns in* `ValOut` *the stored value. Rule* `l2` *causes a lookup to fail if there is no matching entry. Clearly the rules are non-confluent in the theoretical operational semantics, since they rely on rule ordering to give the intended behaviour.* □

The refined operational semantics also allows us to create more efficient programs and/or have a better idea regarding their time complexity.

Example 16 *Consider the following implementation of Fibonacci numbers,* `fib(N,F)`, *which holds if* F *is the* N^{th} *Fibonacci number:*

```
f1 @ fib(N,F) <=> 1 >= N | F = 1.
f2 @ fib(N,F0) \ fib(N,F) <=> N >= 2 | F = F0.
f3 @ fib(N,F) ==> N >= 2 | fib(N-2, F1), fib(N-1,F2), F = F1 + F2.
```

[1] Unlike Prolog, function call "$m - n$" is evaluated as an integer subtraction.

The program is confluent in the theoretical operational semantics which, as we will see later, means it is also confluent in the refined operational semantics. Under the refined operational semantics it has linear complexity, while swapping rules `f2` *and* `f3` *leads to exponential complexity. Since in the theoretical operational semantics both versions are equivalent, complexity is at best exponential.* □

We believe that CHRs under the refined operational semantics provide a powerful and elegant language suitable for general purpose computing. However, to make use of this language, authors need support to ensure their code is confluent within this context. In order to do this, we first provide a formal definition of the refined operational semantics of CHRs as implemented in logic programming systems. We then provide theoretical results linking the refined and theoretical operational semantics. Essentially, these results ensure that if a program is confluent and terminating under the theoretical semantics, it is also confluent and terminating under the refined semantics.

For the rest of this thesis, we sometimes use symbol 'ω_t' to represent the "theoretical semantics". Likewise, we use symbol 'ω_r' to represent the "refined semantics".

3.2 The Theoretical Operational Semantics ω_t

Constraints can be divided into either *CHR* constraints or *built-in* constraints in some constraint domain $\mathcal{D}$. Decisions about rule matchings will rely on the underlying solver proving that the current constraint store for the underlying solver entails a *guard* (a conjunction of built-in constraints). We will assume the solver supports (at least) equality.

As was introduced in Chapter 2, there are three types of rules: simplification, propagation and simpagation. For simplicity, we consider both simplification and propagation rules as special cases of a simpagation rules. The general form of a *simpagation* rule is:

$$r \ @ \ H_1 \setminus H_2 \iff g \mid B$$

where r is the rule name, H_1 and H_2 are sequences of CHR constraints, g is a sequence of built-in constraints, and B is a sequence of constraints. At least one of H_1 and H_2 must be non-empty. Finally, a CHR program P is a sequence of rules.

We shall sometimes treat multisets as sequences, in which case we nondeterministically choose an order for the objects in the multiset.

Given a CHR program P, we will be interested in numbering the occurrences of each CHR constraint predicate p appearing in the head of the rule. We number the occurrences following the top-down rule order and right-to-left constraint order. The latter is aimed at ordering first the constraints after the backslash (\) and then those before it, since this is more efficient in general.

Example 17 *The following shows the* `gcd` *CHR program of Example 14, written using simpagation rules and all occurrences numbered:*

```
gcd1 @ []       \ gcd(0)_1 <=> true   | true.
gcd2 @ gcd(N)_3 \ gcd(M)_2 <=> M ≥ N | gcd(M-N).
```

□

3.2.1 The ω_t Semantics

Several versions of the theoretical operational semantics have already appeared in the literature, e.g. [1, 32], essentially as a multiset rewriting semantics. This section presents our variation,[2] which subsumes previous versions, and is close enough to our refined operational semantics to make proofs simple.

Firstly we define *numbered* constraints.

Definition 1 (Numbered Constraints) *A* numbered constraint *is a constraint c paired with an integer i. We write $c\#i$ to indicate a numbered constraint.* □

Sometimes we refer to i as the *identifier* (or simply ID) of the numbered constraint. This numbering serves to differentiate among copies of the same constraint.

Now we define an *execution state*, as follows.

Definition 2 (Execution State) *An* execution state *is a tuple of the form $\langle G, S, B, T\rangle_n^{\mathcal{V}}$ where G is a multiset (repeats are allowed) of constraints, S is a set of numbered constraints, B is a conjunction of built-in constraints, T is a set of sequences of integers, $\mathcal{V}$ is the set of variables and n is an integer. Throughout this thesis we use symbol 'σ' to represent an execution state.* □

We call G the *goal*, which contains all constraints to be executed. The CHR constraint *store* S is the set[3] of *numbered* CHR constraints that can be matched with rules in the program P. For convenience we introduce functions $cons(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets of numbered CHR constraints in the obvious manner.

The *built-in constraint store* B contains any built-in constraint that has been passed to the built-in solver. Since we will usually have no information about the internal representation of B, we treat it as a conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints which fired a rule, and the name of the rule itself (which may be represented as a unique integer, but typically we just use the name of the rule itself). This is necessary to prevent trivial nontermination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints

[2] A brief comparison between this and previous formalisations of the semantics can be found later in Section 3.6.

[3] Sometimes we treat the store as a multiset.

have not been used to fire the rule before. The set $\mathcal{V}$ contains all variables that appeared in the initial goal. Its purpose will become clear in Section 3.4 where we define the declarative semantics of execution states. Throughout this thesis we will usually omit $\mathcal{V}$ unless we require it to be explicitly shown. Finally, the counter n represents the next free integer which can be used to number a CHR constraint.

We define an *initial state* as follows.

Definition 3 (Initial State) *Given a goal G, which is a multiset of constraints, the* initial state *with respect to G is $\langle G, \emptyset, true, \emptyset \rangle_1^{vars(G)}$.* □

A built-in solver determines if a logical formula F holds with respect to a constraint theory $\mathcal{D}$.

Definition 4 *We model the built-in solver S by the test $(\mathcal{D} \models_{\mathcal{S}} F)$, where F is an arbitrary logical formula. We require that $\mathcal{D} \models_{\mathcal{S}} F$ satisfies (at least) the following conditions for all formulae G and H:*

1. *correctness: If $\mathcal{D} \models_{\mathcal{S}} F$ holds then $\mathcal{D} \models F$ holds.*
2. *monotonic: if $\mathcal{D} \models_{\mathcal{S}} (F \rightarrow G)$ holds then $\mathcal{D} \models_{\mathcal{S}} (F \wedge H \rightarrow G)$ holds.*

We also define the test $(\mathcal{D} \not\models_{\mathcal{S}} F)$, which holds iff $\mathcal{D} \models_{\mathcal{S}} F$ does not hold. □

A built-in solver S is *complete* if the test $(\mathcal{D} \models_{\mathcal{S}} F)$ is equivalent to $(\mathcal{D} \models F)$ for all formulae F, otherwise S is *incomplete*. For a complete solver, the test $(\mathcal{D} \not\models_{\mathcal{S}} F)$ is equivalent to $(\mathcal{D} \models \neg F)$.

The theoretical operational semantics ω_t is based on the following three transitions which map execution states to execution states:

Definition 5 (Theoretical Operational Semantics)
1. Solve

$$\langle \{c\} \uplus G, S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle G, S, c \wedge B, T \rangle_n^{\mathcal{V}}$$

where c is a built-in constraint.
2. Introduce

$$\langle \{c\} \uplus G, S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle G, \{c\#n\} \uplus S, B, T \rangle_{(n+1)}^{\mathcal{V}}$$

where c is a CHR constraint.
3. Apply

$$\langle G, H_1 \uplus H_2 \uplus S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle \theta(C) \uplus G, H_1 \uplus S, B, T' \rangle_n^{\mathcal{V}}$$

where there exists a (renamed apart) rule in P of the form

$$r \ @ \ H_1' \setminus H_2' \iff g \mid C$$

and the matching substitution θ is such that

$$\begin{cases} cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [r]\}$.[4] □

The **Solve** transition adds a new built-in constraint from goal G to the built-in store B. The **Introduce** transition adds a new numbered CHR constraint to the CHR store S. The **Apply** transition chooses a rule from the program such that matching constraints exist in the CHR store S, and the guard is entailed by the built-in store B, and fires it. For readability, we usually apply the resulting substitution θ to all relevant fields in the execution state, i.e. G, S and B. This does not affect the meaning of the execution state, or its transition applicability, but it helps remove the build-up of too many variables and constraints.

The theoretical operational semantics states that given a goal G, we nondeterministically apply the transitions from Definition 5 until a *final state* is reached. We define a final state as follows.

Definition 6 (Final States) *An execution state $\sigma = \langle G, S, B, T \rangle_n$ is a* final state *if either no transition defined in Definition 5 is applicable to σ, or $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset} B$ holds (often we simply use $false$ to represent such a state).* □

The sequence of execution states generated by continuously applying transition steps is called a *derivation*, which is formally defined as follows.

Definition 7 (Derivation) *A derivation D is a non-empty (but possibly infinite) sequence of execution states $D = [\sigma_0, \sigma_1, \sigma_2, ...]$ such that σ_{i+1} is the result of applying a transition from Definition 5 to execution state σ_i for all consecutive states σ_i and σ_{i+1} in D.* □

Usually we write $\sigma_0 \rightarrowtail \sigma_1 \rightarrowtail \sigma_2 \rightarrowtail ...$ instead of $[\sigma_0, \sigma_1, \sigma_2, ...]$ to denote a derivation, and the length of the derivation is the length of the sequence less one. Sometimes we use the notation $D = D_0 +\!\!+ D_1$ to represent a partition of derivation D.

Example 18 *Figure 3.1 is a (terminating) derivation under ω_t for the query* `gcd(6), gcd(9)` *executed on the* `gcd` *program in Example 17. For brevity, B, T and $\mathcal{V}$ have been removed from each tuple. No more transitions on state $\langle \emptyset, \{\texttt{gcd}(3)\#3\} \rangle_6$ are possible, so this is the final state.* □

[4] Note in practice we only need to keep track of tuples where H_2 is empty, since otherwise these CHR constraints are being deleted and the firing can not reoccur.

$$
\begin{array}{lll}
 & & \langle \{\texttt{gcd}(6), \texttt{gcd}(9)\}, \emptyset \rangle_1 \quad (1) \\
 & \rightarrowtail_{introduce} & \langle \{\texttt{gcd}(9)\}, \{\texttt{gcd}(6)\#1\} \rangle_2 \quad (2) \\
 & \rightarrowtail_{introduce} & \langle \emptyset, \{\texttt{gcd}(6)\#1, \texttt{gcd}(9)\#2\} \rangle_3 \quad (3) \\
(\texttt{gcd2} \quad N = 6 \wedge M = 9) & \rightarrowtail_{apply} & \langle \{\texttt{gcd}(3)\}, \{\texttt{gcd}(6)\#1\} \rangle_3 \quad (4) \\
 & \rightarrowtail_{introduce} & \langle \emptyset, \{\texttt{gcd}(6)\#1, \texttt{gcd}(3)\#3\} \rangle_4 \quad (5) \\
(\texttt{gcd2} \quad N = 3 \wedge M = 6) & \rightarrowtail_{apply} & \langle \{\texttt{gcd}(3)\}, \{\texttt{gcd}(3)\#3\} \rangle_4 \quad (6) \\
 & \rightarrowtail_{introduce} & \langle \emptyset, \{\texttt{gcd}(3)\#3, \texttt{gcd}(3)\#4\} \rangle_5 \quad (7) \\
(\texttt{gcd2} \quad N = 3 \wedge M = 3) & \rightarrowtail_{apply} & \langle \{\texttt{gcd}(0)\}, \{\texttt{gcd}(3)\#3\} \rangle_5 \quad (8) \\
 & \rightarrowtail_{introduce} & \langle \emptyset, \{\texttt{gcd}(3)\#3, \texttt{gcd}(0)\#5\} \rangle_6 \quad (9) \\
(\texttt{gcd1}) & \rightarrowtail_{apply} & \langle \emptyset, \{\texttt{gcd}(3)\#3\} \rangle_6 \quad (10)
\end{array}
$$

Figure 3.1: ω_t derivation for `gcd`.

3.3 The Refined Operational Semantics ω_r

The *refined* operational semantics establishes an order for the constraints in G. As a result, we are no longer free to pick any constraint from G to either **Solve** or **Introduce** into the store. It also treats CHR constraints as procedure calls: each newly added CHR constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires other CHR constraints might be executed and, when they finish, the execution returns to finding rules for the current constraint. Not surprisingly, this approach is used exactly because it corresponds closely to that of the language we compile to.

Formally, the execution state of the refined semantics is the tuple

$$\langle A, S, B, T \rangle_n^{\mathcal{V}}$$

where S, B, T, $\mathcal{V}$ and n, representing the CHR store, built-in store, propagation history, initial variables and next free identity number respectively, are exactly as with Definition 2. The *execution stack* A is a sequence of constraints, numbered CHR constraints and *active* CHR constraints, with a strict ordering in which only the top-most constraint is considered for execution.[5] We now define *active constraints*, which represent a specific call to a CHR constraint.

Definition 8 (Active Constraints) *An* active *constraint $c\#i : j$ is a numbered CHR constraint $c\#i$ associated with an integer j which represents the* occurrence *of predicate c in P the constraint $c\#i$ is allowed to match with.* □

Unlike in the theoretical operational semantics, a numbered constraint may simultaneously appear in both the execution stack A and the store S.

[5]The execution stack is analogous to a call-stack in other programming languages, e.g. Prolog etc.

Given initial goal G, the initial state is as before, i.e. of the form

$$\langle G, \emptyset, true, \emptyset \rangle_1^{vars(G)}$$

except this time G is an ordered sequence, rather than a constraint multiset. Just as with the theoretical operational semantics, execution proceeds by exhaustively applying transitions to the initial execution state until the built-in solver state is unsatisfiable or no transitions are applicable.

The refined operational semantics treats CHR constraints with only *fixed variables* as a special case. We formally define *fixed variable* as follows.

Definition 9 (Fixed) *Let B be a built-in store, then $v \in fixed(B)$ if*

$$\mathcal{D} \models_{\mathcal{S}} \forall v \forall \rho(v)(\bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v))$$

for arbitrary renaming ρ. □

Informally, a variable which can only take one value to satisfy B is fixed. We say a constraint c is fixed if $vars(c) \subseteq fixed(B)$.

When a built-in constraint is added to the built-in store, the refined semantics *wakes up* a subset of the CHR store to be reconsidered for execution. The exact subset is left open, however it must satisfy the conditions of a *wakeup policy*, which is defined as follows.

Definition 10 (Wakeup Policy) *Let S be a CHR store, c a built-in constraint and B a built-in store, then a* wakeup policy *is a function $wakeup\_policy(S, c, B) = S_1$ where S_1 is a finite multiset such that for all $s \in S_1$ we have that $s \in S$, and S_1 satisfies the following further conditions:*

1. lower bound*: For all $M = H_1 +\!\!+ H_2 \subseteq S$ such that there exists a rule*

$$r \ @ \ H_1' \setminus H_2' \Longleftrightarrow g \mid C$$

and a substitution θ such that

$$\begin{cases} cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ \mathcal{D} \not\models_{\mathcal{S}} (B \rightarrow \exists_r(\theta \wedge g)) \\ \mathcal{D} \models_{\mathcal{S}} (B \wedge c \rightarrow \exists_r(\theta \wedge g)) \end{cases}$$

then $M \cap S_1 \neq \emptyset$

2. upper bound*: If $m \in S_1$ then $vars(m) \not\subseteq fixed(B)$.*

□

Each implementation of the refined semantics provides its own wakeup policy. The *lower bound* ensures that S_1 is (at least) the minimum subset of the store that actually needs to be reconsidered thanks to the addition of c. The *upper bound* ensures that S_1 contains no fixed constraints. This will ensure that fixed constraints have a more deterministic behaviour, which is essential for confluence (see Chapter 6). Note that the definition of a wakeup policy also allows the set S_1 to contain multiple (redundant) copies of constraints in S.[6]

We can now define the refined operational semantics of CHRs.

Definition 11 (Refined Operational Semantics)
1. Solve

$$\langle [c|A], S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle wakeup\_policy(S, c, B) +\!\!+ A, S, c \wedge B, T\rangle_n^{\mathcal{V}}$$

where c is a built-in constraint.
2. Activate

$$\langle [c|A], S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{(n+1)}^{\mathcal{V}}$$

where c is a CHR constraint (which has never been active).
3. Reactivate

$$\langle [c\#i|A], S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle [c\#i : 1|A], S, B, T\rangle_n^{\mathcal{V}}$$

where c is a CHR constraint (re-added to A by **Solve** but not yet active).
4. Drop

$$\langle [c\#i : j|A], S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle A, S, B, T\rangle_n^{\mathcal{V}}$$

where $c\#i : j$ is an active constraint and there is no such occurrence j in P (all existing ones have already been tried thanks to transition 7).
5. Simplify

$$\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \theta(C) +\!\!+ A, H_1 \uplus S, B, T'\rangle_n^{\mathcal{V}}$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

and there exists a matching substitution θ such that

$$\begin{cases} c = \theta(d_j) \\ cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ cons(H_3) = \theta(H_3') \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

[6]This models the behaviour of a CHR constraint waking up more than once when a built-in constraint is added to the store, which may happen in practice.

In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.[7]

6. Propagate

$$\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \theta(C) +\!\!+ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, B, T'\rangle_n^{\mathcal{V}}$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1', d_j, H_2' \setminus H_3' \iff g \mid C$$

and the matching substitution θ is such that

$$\begin{cases} c = \theta(d_j) \\ cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ cons(H_3) = \theta(H_3') \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r]\}$.

The role of the propagation histories T and T' is exactly the same as with the theoretical operational semantics, ω_t.

7. Default

$$\langle [c\#i : j|A], S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle [c\#i : j+1|A], S, B, T\rangle_n^{\mathcal{V}}$$

if the current state cannot fire any other transition. □

Some of the transitions for the refined operational semantics are analogous to transitions under the theoretical semantics. For example, the refined **Solve** transition corresponds to the theoretical **Solve** transition, as they both introduce a new built-in constraint into store B. Likewise, **Active** corresponds to **Introduce** (add a new CHR constraint to the store), and **Simplify** and **Propagate** correspond to **Apply** (fires a rule on some constraints in the store). This correspondence is not accidental, and later in this chapter we formally define a mapping between the semantics.

The main difference between the refined and theoretical semantics is the presence (and behaviour) of *active* constraints. A rule r can only fire (via **Simplify** or **Propagate**) if the occurrence number of the current active constraint (on top of the execution stack) appears in the head of rule r. An active constraint is only allowed to match against the constraint in the head of r that shares the same occurrence number.

Example 19 *For example, an active constraint* $\texttt{leq}(A, B)\#1 : 2$ *(with occurrence number* 2*) is only allowed to match against* $\texttt{leq}(Y, X)_2$ *(also with occurrence number* 2*) in the following rule.*

[7] As with the theoretical semantics, it is not necessary to check the history if H_2 is not empty. We include the check anyway to simplify our proofs later in this chapter.

```
leq(X,Y)1, leq(Y,X)2 <=> X = Y.
```

□

This is very different to the theoretical semantics, where a rule is free to fire on any subset of the current CHR store.

If the current active constraint cannot match against the associated rule, e.g. if all matchings have already been tried, then the active constraint "moves" to the next occurrence via the **Default** transition. This ensures that, assuming termination, all occurrences will eventually be tried. When there are no more occurrences to check, i.e. the occurrence number associated to an active constraint does not appear in P, then we can apply **Drop**. This pops off the current active constraint, but does not remove anything from the store.

Initially there are no active constraints in the goal, but we can turn a non-active constraint into an active constraint via the **Activate** transition. The **Activate** transition associates the first occurrence (defined as occurrence number 1) with the constraint, and adds a copy of the constraint to the CHR constraint store (just like **Introduce** under the theoretical semantics).

The **Solve** transition is also handled differently. After applying **Solve**, a subset of the CHR store (defined by the wakeup policy) is appended to the front of the execution stack. The intention is that these constraints will eventually become active again (via the **Reactivate** transition), and will reconsider all rules in the program P. These active constraints may fire against more rules than before, because the addition of the new built-in constraint c may mean the underlying solver can prove more guards hold than before.

Example 20 *Consider the following rule from the* `leq` *program in Example 1.*

```
leq(X,X)1 <=> true.
```

Assume the built-in store B is empty (i.e. $B = true$), then a constraint $\texttt{leq}(J, K)$ *where J and K are distinct variables cannot fire this rule because $\mathcal{D} \models true \rightarrow \exists X((J = X \wedge K = X) \wedge true)$ is not equivalent to true (i.e. the guard does not hold). However, if a new constraint $J = K$ were to be added into the built-in store via a **Solve** transition, then $\mathcal{D} \models (J = K) \rightarrow \exists X(((J = X \wedge K = X) \wedge true)$ always holds, thus the rule can now fire. Here, we are assuming that the built-in solver is complete.*

Under the refined semantics, the constraint $\texttt{leq}(J, K)$ *will be copied to the execution stack during the **Solve** transition. Therefore eventually (assuming termination), the constraint will be **Reactivate**d, and fire the rule.* □

The refined operational semantics constrains the values a wakeup policy is allowed to return. At the very minimum, a wakeup policy must contain a CHR constraint from every new matching that is possible thanks to the addition of c into the built-in store (the *lower bound* condition). At most, a wakeup policy returns all non-ground constraints currently in the CHR store (the *upper bound* condition). The exact implementations of the wakeup policy is left to the implementation.

We now present some examples of derivations under the refined operational semantics.

3.3.1 Extended Example: `leq`

In this section we present an extended example of the refined operational semantics. Consider the `leq` program from Example 1, which defines a CHR constraint $\texttt{leq}(X, Y)$ representing the ordering relation $X \leq Y$. Unlike with the `gcd` constraint, arguments for `leq` may be solver variables, hence we may need to wakeup constraints if/when the built-in store changes. We will assume that the built-in solver is complete.

The rules defining the `leq` constraint are given below.

```
reflexivity   @ leq(X,X)₁                <=> true.
antisymmetry  @ leq(X,Y)₂, leq(Y,X)₃     <=> X = Y.
idempotence   @ leq(X,Y)₅ \ leq(X,Y)₄    <=> true.
transitivity  @ leq(X,Y)₆, leq(Y,Z)₇     <=> leq(X,Z).
```

Each occurrence in the program has been labelled a number between 1..7. Active constraints are checked against occurrences in this order.

Our initial query is $\texttt{leq}(A,B),\texttt{leq}(C,A),\texttt{leq}(B,C)$ (the same as that from Example 2), which represents the ordering relations $A \leq B \wedge C \leq A \wedge B \leq C$. The initial state for this goal is

$$\langle [\texttt{leq}(A,B), \texttt{leq}(C,A), \texttt{leq}(B,C)], \emptyset, true, \emptyset \rangle_1^{\{A,B,C\}}$$

As this is an initial state, the store S, built-in store B and propagation history T are all empty. Now we exhaustively apply ω_r transitions until either failure is reached, or no more transitions can be applied. For brevity, the propagation history T and variables $\mathcal{V}$ will be omitted for the rest of this example.

The very first transition activates the left-most CHR constraint on the execution stack.

$$\langle [\texttt{leq}(A,B)\#1 : 1, \texttt{leq}(C,A), \texttt{leq}(B,C)], \{\texttt{leq}(A,B)\#1\}, true \rangle_2 \overset{\rightarrowtail_{activate}}{}$$

Constraint $\texttt{leq}(A,B)$ is now activated with new ID number 1. The constraints now also appears in the CHR store.

The activated constraint is now ready to visit each occurrence, and check if the corresponding rule can fire. Since the arguments to the `leq` constraint, namely variables A and B, are distinct, the first rule cannot fire, since the guard $A = B$ is not satisfied (just as with Example 20 before). All other rules require at least two `leq` constraints to be present in the CHR store. Currently there is only one, therefore the only applicable transition is **Default**.

$$\langle [\texttt{leq}(A,B)\#1 : 8, \texttt{leq}(C,A), \texttt{leq}(B,C)], \{\texttt{leq}(A,B)\#1\}, true \rangle_2 \overset{\rightarrowtail^{\times 7}_{default}}{}$$

As there is no occurrence number 8, the only applicable transition is **Drop**, which simply removes the active constraint from the execution stack.

$$\rightarrowtail_{drop}$$
$$\langle [\texttt{leq}(C,A), \texttt{leq}(B,C)], \{\texttt{leq}(A,B)\#1\}, true\rangle_2$$

The constraint `leq(`A`,`B`)` has finished execution, so the next left-most constraint `leq(`C`,`A`)` must now be activated by the **Activate** transition. None of the rules for occurrences 1...5 can fire on this new active constraint, hence the **Default** transition is applied 5 times

$$\rightarrowtail_{activate}\rightarrowtail_{default}^{\times 5}$$
$$\langle [\texttt{leq}(C,A)\#2:6, \texttt{leq}(B,C)], \{\texttt{leq}(C,A)\#2, \texttt{leq}(A,B)\#1\}, true\rangle_3$$

The active constraint is currently at occurrence 6.

Now we find that the **Propagate** transition can be applied with the matching substitution θ as $(X = C \wedge Y = A \wedge Y = A \wedge Z = B)$. A simple check verifies all of the conditions for **Propagate** are satisfied, namely,

- the guard is satisfied, i.e. $\mathcal{D} \models true \rightarrow \exists X \exists Y \exists Z(\theta \wedge true)$ holds; and
- the application of the rule is not blocked by the propagation history, i.e. $[1, 2, r4] \notin T$ (as T is currently empty).

The resulting state after **Propagate** is:

$$\langle [\texttt{leq}(C,B), \texttt{leq}(C,A)\#2:6, \texttt{leq}(B,C)], \{\texttt{leq}(C,A)\#2, \texttt{leq}(A,B)\#1\}, true\rangle_3$$

The top of the execution stack is the constraint `leq(`C`,`B`)` which is **Activate**d, and assigned the ID 3. The new active constraint does not match any occurrence, so **Default** is applied 7 times, then **Drop**.

$$\rightarrowtail_{activate}\rightarrowtail_{default}^{\times 7}\rightarrowtail_{drop}$$
$$\langle [\texttt{leq}(C,A)\#2:6, \texttt{leq}(B,C)], \{\texttt{leq}(C,B)\#3, \texttt{leq}(C,A)\#2, \texttt{leq}(A,B)\#1\}\rangle_4$$

Now the active constraint $\texttt{leq}(C,A)\#2:6$ is on top of the stack again.

It seems that **Propagate** could be applied again (just as before), however there now exists an entry $[r4, 1, 2] \in T$ which prevents this from happening. There is no other choice but to apply **Default**, and ultimately **Drop**.

$$\rightarrowtail_{default}^{\times 2}\rightarrowtail_{drop}$$
$$\langle [\texttt{leq}(B,C)], \{\texttt{leq}(C,B)\#3, \texttt{leq}(C,A)\#2, \texttt{leq}(A,B)\#1\}, true\rangle_4$$

Now `leq(`B`,`C`)` is activated (with new ID 4), but no occurrence can fire until occurrence 2.

$$\rightarrowtail_{activate}\rightarrowtail_{default}$$
$$\langle [\texttt{leq}(B,C)\#4:2], \{\texttt{leq}(B,C)\#4, \texttt{leq}(C,B)\#3, \texttt{leq}(C,A)\#2, \texttt{leq}(A,B)\#1\}\rangle_5$$

Now we find **Simplify** can be applied on the active constraint and $\texttt{leq}(C,B)$ from the store, with the matching substitution θ as $(Y = B \wedge X = C \wedge X' = C \wedge Y' = B)$. The guard is satisfied and $[4, 3, r2] \notin T$. Both the active constraint $\texttt{leq}(B,C)$ and constraint $\texttt{leq}(C,B)$ are deleted, and the built-in constraint $C = B$ is pushed onto the top of execution stack.

$$\rightarrowtail_{simplify} \langle [C = B], \{\texttt{leq}(C, A)\#2, \texttt{leq}(A, B)\#1\}, true\rangle_5$$

Next **Solve** is applied to the built-in constraint on the execution stack. This moves the constraint $C = B$ to the built-in store. We also assume a very simple wakeup policy: all non-fixed CHR constraints from the store will be woken up. The order that these constraints are added to the stack is not specified by the refined semantics, so we pick the order arbitrarily. For simplicity, we also apply constraint $C = B$ to the CHR constraints, replacing variable B with C.

$$\rightarrowtail_{solve} \langle [\texttt{leq}(C, A)\#2, \texttt{leq}(A, C)\#1], \{\texttt{leq}(C, A)\#2, \texttt{leq}(A, C)\#1\}, C = B\rangle_5$$

We **Reactivate** constraint $\texttt{leq}(C,A)$. No rule fires until occurrence 2.

$$\rightarrowtail_{reactivate} \rightarrowtail_{default} \langle [\texttt{leq}(C, A)\#2 : 2, \texttt{leq}(A, C)\#1], \{\texttt{leq}(C, A)\#2, \texttt{leq}(A, C)\#1\}, C = B\rangle_5$$

We apply **Simplify** with the matching substitution $\theta = (X = A \wedge Y = C)$. The guard is satisfied and $[1, 2, r2] \notin T$. Both CHR constraints are deleted and the body constraint $C = A$ is added to the execution stack.

$$\rightarrowtail_{simplify} \langle [C = A, \texttt{leq}(A, C)\#1], \emptyset, C = B\rangle$$

We apply **Solve** to the equation.

$$\rightarrowtail_{solve} \langle [\texttt{leq}(A, C)\#1], \emptyset, C = A \wedge C = B\rangle$$

Finally, we **Reactivate** the remaining CHR constraint. Since this constraint no longer appears in the CHR store, no rule can fire. Therefore the constraint is eventually **Drop**ped.

$$\rightarrowtail_{reactivate} \rightarrowtail_{default}^{\times 7} \rightarrowtail_{drop} \langle [], \emptyset, C = A \wedge C = B\rangle$$

Since the execution stack is empty, we have reached a final state. Therefore, the solution to the original query $(A \leq B \wedge C \leq A \wedge B \leq C)$ is $(C = A \wedge C = B)$, as expected.

3.3.2 Small Example: `gcd`

Figure 3.2 shows the derivation under ω_r semantics for the `gcd` program in Example 17 and the goal `gcd(6),gcd(9)`. For brevity B, T and $\mathcal{V}$ have been eliminated.

$$
\begin{array}{ll}
 & \langle[\texttt{gcd}(6), \texttt{gcd}(9)], \emptyset\rangle_1 \quad (1)\\
\rightarrowtail_{activate} & \langle[\texttt{gcd}(6)\#1:1, \texttt{gcd}(9)], \{\texttt{gcd}(6)\#1\}\rangle_2 \quad (2)\\
\rightarrowtail_{default}^{\times 3} & \langle[\texttt{gcd}(6)\#1:4, \texttt{gcd}(9)], \{\texttt{gcd}(6)\#1\}\rangle_2 \quad (2)\\
\rightarrowtail_{drop} & \langle[\texttt{gcd}(9)], \{\texttt{gcd}(6)\#1\}\rangle_2 \quad (2)\\
\rightarrowtail_{activate}\rightarrowtail_{default} & \langle[\texttt{gcd}(9)\#2:2], \{\texttt{gcd}(9)\#2, \texttt{gcd}(6)\#1\}\rangle_3 \quad (3)\\
\rightarrowtail_{simplify} & \langle[\texttt{gcd}(3)], \{\texttt{gcd}(6)\#1\}\rangle_3 \quad (4)\\
\rightarrowtail_{activate}\rightarrowtail_{default}^{\times 2} & \langle[\texttt{gcd}(3)\#3:3], \{\texttt{gcd}(3)\#3, \texttt{gcd}(6)\#1\}\rangle_3 \quad (5)\\
\rightarrowtail_{propagate} & \langle[\texttt{gcd}(3), \texttt{gcd}(3)\#3:3], \{\texttt{gcd}(3)\#3\}\rangle_4 \quad (6)\\
\rightarrowtail_{activate}\rightarrowtail_{default} & \langle[\texttt{gcd}(3)\#4:2, \texttt{gcd}(3)\#3:3], \{\texttt{gcd}(3)\#4, \texttt{gcd}(3)\#3\}\rangle_5 \quad (7)\\
\rightarrowtail_{simplify} & \langle[\texttt{gcd}(0), \texttt{gcd}(3)\#3:3], \{\texttt{gcd}(3)\#3\}\rangle_5 \quad (8)\\
\rightarrowtail_{activate} & \langle[\texttt{gcd}(0)\#5:1, \texttt{gcd}(3)\#3:3], \{\texttt{gcd}(0)\#5, \texttt{gcd}(3)\#3\}\rangle_6 \quad (9)\\
\rightarrowtail_{simplify} & \langle[\texttt{gcd}(3)\#3:3], \{\texttt{gcd}(3)\#3\}\rangle_6 \quad (10)\\
\rightarrowtail_{default}\rightarrowtail_{drop} & \langle[], \{\texttt{gcd}(3)\#3\}\rangle_6 \quad (10)
\end{array}
$$

Figure 3.2: ω_r derivation for gcd.

3.4 Declarative Semantics

In this section, we establish the declarative semantics (logical interpretation) of execution states and CHR programs. This is essentially a brief overview of the declarative semantics that has appeared in past literature, e.g. in [32].

Firstly we give the definition for the logical interpretation of states under the theoretical operational semantics.

Definition 12 (Logical Interpretation of Execution States) *The logical interpretation of an ω_t execution state σ is given by the function $[\![\sigma]\!]$, which is defined as*

$$
\begin{array}{lll}
[\![\langle G, S, B, T\rangle_i^{\mathcal{V}}]\!] & = \bar{\exists}_{\mathcal{V}}([\![G]\!] \wedge [\![S]\!] \wedge B) & \\
[\![\{c\} \uplus S]\!] & = [\![c]\!] \wedge [\![S]\!] & \\
[\![c\#i]\!] & = c & \\
[\![c]\!] & = c & (c \textit{ not numbered})
\end{array}
$$

□

Basically, the logical interpretation of an execution state is the conjunction of the goal G, store S and built-in store B. The purpose of $\mathcal{V}$ is to indicate which variables are existentially quantified, i.e. variable v will be existentially quantified if $v \notin \mathcal{V}$.

The definition for the logical interpretation of execution states under the refined operational semantics is delayed until later in this chapter, once the relationship between the two semantics has been defined.

We extend the function $[\![]\!]$ to give the logical interpretation of CHR programs.

Definition 13 (Logical Interpretation of CHR programs) *Let P be a CHR program (a sequence of rules), then the logical interpretation $[\![P]\!]$ is defined as*

$$\begin{array}{ll} [\![[]]\!] & = true \\ [\![[r|P]]\!] & = [\![r]\!] \wedge [\![P]\!] \\ [\![(H_1 \setminus H_2 \iff g \mid C)]\!] & = \forall \bar{x}((\exists \bar{y}\ g) \rightarrow \exists \bar{z}([\![H_1]\!] \wedge [\![H_2]\!] \leftrightarrow [\![C]\!] \wedge [\![H_1]\!]))) \end{array}$$

Here, $\bar{x} = vars(H_1 \wedge H_2)$ (variables appearing in the head), $\bar{y} = vars(g) - \bar{x}$ (variables in the guard only) and $\bar{z} = vars(C) - \bar{x}$ (variables appearing in the body only). □

For simplicity, this definition assumes that the guard and body do not share variables that do not appear in the head. If is not the case, then we treat the rule as

$$(H_1 \setminus H_2 \iff g \mid \rho(g \wedge C))$$

where ρ renames all variables $v \in vars(g)$ such that $v \in vars(C)$ and $v \notin vars(H_2 \wedge H_2)$. We refer to this process as *body normalisation.*

The declarative semantics is important for many applications. For example, the declarative semantics of a constraint solver implemented in CHRs usually corresponds with the *constraint theory* $\mathcal{D}$ of that solver.

Example 21 *The following is the (simplified) declarative semantics of the* `leq` *program from Example 1.*

$$\begin{array}{c} \forall X(\texttt{leq}(X,X) \leftrightarrow true) \wedge \\ \forall X \forall Y(\texttt{leq}(X,Y) \wedge \texttt{leq}(Y,X) \leftrightarrow X = Y) \wedge \\ \forall X \forall Y(\texttt{leq}(X,Y) \wedge \texttt{leq}(X,Y) \leftrightarrow \texttt{leq}(X,Y)) \wedge \\ \forall X \forall Y \forall Z(\texttt{leq}(X,Y) \wedge \texttt{leq}(Y,Z) \rightarrow \texttt{leq}(X,Z))) \end{array}$$

Each formula on each line corresponds to a rule from the original program in order. Notice that the first, second and fourth lines are a logical specification of the reflexivity*,* antisymmetric *and* transitivity *properties of a partial order.*

The third rule implements the property of idempotence*, which is really a property of conjunction. In fact this rule is logically redundant. The idempotence rule is included because of practical reasons, since CHRs allow multiple copies of the same rule by default, even though (from a declarative perspective) one copy is enough.* □

It is sometimes the case that the logical interpretation of a program has little to do with its operational behaviour. This is common when the program relies on the refined operational semantics.

Example 22 *For example, consider the declarative meaning of the simple program presented in Example 15.*

$$\begin{array}{c} \forall K \forall V \forall V'(\texttt{entry}(K,V) \wedge \texttt{lookup}(K,V') \leftrightarrow V = V' \wedge \texttt{entry}(K,V)) \wedge \\ \forall X \forall Y(\texttt{lookup}(X,Y) \leftrightarrow false) \end{array}$$

The last rule means all constraints $\texttt{lookup}(X, Y)$ *are logically equivalent to* $false$*, yet calls to* $\texttt{lookup}(X, Y)$ *may not immediately fail (if there is an appropriate entry in the database). This example shows that the operational and declarative semantics of a program may not always coincide.* □

In this case we are using CHRs as a general rule-based rewrite programming language, hence declarative semantics is not important.

3.5 The Relationship Between ω_t and ω_r

Once both semantics are established, we can define an abstraction function α which maps execution states of ω_r to ω_t. Later, we use this abstraction function to prove correctness of the refined semantics ω_r with respect to the theoretical semantics ω_t.

Definition 14 (Correspondence of States) *The abstraction function* α *is defined as*

$$\alpha(\langle A, S, B, T\rangle_n^{\mathcal{V}}) = \langle no\_id(A), S, B, T\rangle_n^{\mathcal{V}}$$

where $no\_id(A) = \{c \mid c \in A \text{ is not of the form } c\#i \text{ or } c\#i : j\}$. □

The abstraction function removes all numbered constraints from the execution stack, and turns the stack into an unordered multiset.

Example 23 *Consider the following* ω_r *state from the* `gcd` *example.*

$$\langle[\texttt{gcd}(0), \texttt{gcd}(3)\#3 : 3], \{\texttt{gcd}(3)\#3\}\rangle_5$$

After applying α *we get*

$$\langle\{\texttt{gcd}(0)\}, \{\texttt{gcd}(3)\#3\}\rangle_5$$

We have simply removed the constraint $\texttt{gcd}(3)\#3 : 3$ *from the stack, as it was identified with number* 3*, and turned the stack (which is a sequence) into a multiset. The rest of the state is unaffected.* □

We now extend α to map a derivation D under ω_r to the corresponding derivation $\alpha(D)$ under ω_t, by mapping each state appropriately and eliminating adjacent equivalent states.

Definition 15 (Correspondence of derivations) *Function* α *is extended to derivations in* ω_r *as follows*

$$\alpha(\sigma_1 \rightarrowtail D) = \begin{array}{ll} \alpha(D) & \text{if } (D = \sigma_2 \rightarrowtail D' \text{ or } D = \sigma_2) \text{ and } \alpha(\sigma_1) = \alpha(\sigma_2) \\ \alpha(\sigma_1) \rightarrowtail \alpha(D) & \text{otherwise} \end{array}$$

□

Note that this definition is just syntactic, and we do not know if the result of apply function α to a ω_r derivation gives a valid ω_t derivation. We rely on the following theorem to show this.

Theorem 2 (Correspondence) *For all ω_r derivations D, $\alpha(D)$ is a ω_t derivation.*

Proof. By induction. We use D_i to represent a derivation of length i, we also let σ_j be the j^{th} state in D_i, so $D_i = \sigma_0 \rightarrowtail \sigma_1 \rightarrowtail ... \rightarrowtail \sigma_i$. Derivations of length $i+1$ must be constructed from derivations of length i.
Base case: Derivations of zero length. Then $\alpha(D_0) = \alpha(\sigma_0)$ is a ω_t derivation of zero length for all D_0.
Induction Step: Assume that for all derivations D_i of length i, $\alpha(D_i)$ is a ω_t derivation. Let D_{i+1} be a ω_r derivation of length $i+1$ constructed from a derivation D_i by applying a ω_r transition to σ_i (the last state in D_i). We show that $\alpha(D_{i+1})$ is a ω_t derivation.

Let σ_{i+1} be the last state in D_{i+1}, we factor out two possible relationships between $\alpha(\sigma_i)$ and $\alpha(\sigma_{i+1})$ and show that these imply $\alpha(D_{i+1})$ is also a ω_t derivation.

- Case 1: $\alpha(\sigma_{i+1}) = \alpha(\sigma_i)$, then

$$\begin{array}{ll} \alpha(D_{i+1}) & = \\ \alpha(D_i \rightarrowtail \sigma_{i+1}) & = \\ \alpha(D_i) & \end{array}$$

  Hence $\alpha(D_{i+1})$ is also a ω_t derivation; or

- Case 2: $\alpha(\sigma_i) \rightarrowtail_{\omega_t} \alpha(\sigma_{i+1})$ holds for some ω_t transition $\rightarrowtail_{\omega_t}$, then

$$\begin{array}{ll} \alpha(D_{i+1}) & = \\ \alpha(D_i \rightarrowtail_{\omega_r} \sigma_{i+1}) & = \\ \alpha(D_i) \rightarrowtail_{\omega_t} \alpha(\sigma_{i+1}) & \end{array}$$

  This means $\alpha(D_{i+1})$ is constructed from ω_t derivation $\alpha(D_i)$ by applying a ω_t transition to the last state $\alpha(\sigma_i)$, hence $\alpha(\sigma_{i+1})$ is a ω_t derivation by definition.

It remains to be shown that if $\sigma_i \rightarrowtail \sigma_{i+1}$ under ω_r, then either Case 1 or Case 2 holds for $\alpha(\sigma_i)$ and $\alpha(\sigma_{i+1})$. For this we consider all possibilities for the ω_r transition from σ_i to σ_{i+1}.
CASE **Solve**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c|A], S, B, T\rangle_n \rightarrowtail_{solve} \langle wakeup\_policy(S, c, B) +\!\!+ A, S, c \wedge B, T\rangle_n$$

Where c is some built-in constraint. Then $\alpha(\sigma_i) = \langle \{c\} \uplus no\_id(A), S, B, T\rangle_n$. We can apply the ω_t version of **Solve** to $\alpha(\sigma_i)$.

$$\langle \{c\} \uplus no\_id(A), S, B, T\rangle_n \rightarrowtail_{solve} \langle no\_id(A), S, c \wedge B, T\rangle_n$$

Now $\langle no\_id(A), S, c \wedge B, T\rangle_n = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{\omega_t} \alpha(\sigma_{i+1})$ where the transition is ω_t **Solve**. Hence Case 2 above is satisfied.
CASE **Activate**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c|A], S, B, T\rangle_n \rightarrowtail_{activate} \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$$

Where c is some CHR constraint (not yet numbered). Then $\alpha(\sigma_i) = \langle \{c\} \uplus no\_id(A), S, B, T\rangle_n$. We can apply the ω_t **Introduce** to $\alpha(\sigma_i)$.

$$\langle \{c\} \uplus no\_id(A), S, B, T\rangle_n \rightarrowtail_{introduce} \langle no\_id(A), \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$$

Now $\langle no\_id(A), \{c\#n\} \uplus S, B, T\rangle_{(n+1)} = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{introduce} \alpha(\sigma_{i+1})$ where the ω_t transition is **Introduce**. Hence Case 2 above is satisfied.
CASE **Reactivate**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail_{reactivate} \langle [c\#i : 1|A], S, B, T\rangle_n$$

Then $\alpha(\sigma_i) = \alpha(\sigma_{i+1}) = \langle no\_id(A), S, B, T\rangle_n$. Hence Case 1 above is satisfied.
CASE **Drop**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail_{drop} \langle A, S, B, T\rangle_n$$

Then $\alpha(\sigma_i) = \alpha(\sigma_{i+1}) = \langle no\_id(A), S, B, T\rangle_n$. Hence Case 1 above is satisfied.
CASE **Simplify**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{simplify} \langle \theta(C) +\!\!+ A, H_1 \uplus S, B, T'\rangle_n$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

and the matching substitution θ is such that $c = \theta(d_j)$, $cons(H_1) = \theta(H_1')$, $cons(H_2) = \theta(H_2')$, $cons(H_3) = \theta(H_3')$, and $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T$.

Then $\alpha(\sigma_i) = \langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n$. We show that the ω_t **Apply** transition is applicable to $\alpha(\sigma_i)$, namely

- There exists a (renamed apart) rule in P of the form

  $$r'' \ @ \ H_1'' \setminus H_2'' \iff g'' \mid C''$$

  This is satisfied by $r = r''$, $H_1'' = H_1'$, $H_2'' = (H_2', d_j, H_3')$, $g'' = g$ and $C'' = C$. In other words, the same rule as above.

- There exists a matching substitution θ'' such that $cons(H_1) = \theta''(H_1'')$ and $cons(H_2, c, H_3) = \theta''(H_2'')$. This is satisfied by $\theta'' = \theta$, the same matching substitution from above.

- The guard g'' is satisfied, i.e. $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_{r''}(\theta'' \wedge g'')$. This is satisfied because $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \bar{\exists}_r(\theta \wedge g)$ holds, and $r'' = r$, $\theta'' = \theta$ and $g'' = g$.

- The tuple $id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T$. This is directly satisfied from above.

Hence

$$\langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T \rangle_n \rightarrowtail_{apply} \langle \theta(C) \uplus no\_id(A), H_1 \uplus S, B, T' \rangle_n$$

Now $\langle \theta(C) \uplus no\_id(A), H_1 \uplus S, B, T' \rangle_n = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{apply} \alpha(\sigma_{i+1})$ where the ω_t transition is **Apply**. Hence Case 2 above is satisfied.
CASE **Propagate**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\begin{array}{l}\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T \rangle_n \rightarrowtail_{propagate} \\ \qquad \langle \theta(C) +\!\!+ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_3 \uplus S, B, T' \rangle_n\end{array}$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$$

and the matching substitution θ is such that $c = \theta(d_j)$, $cons(H_1) = \theta(H'_1)$, $cons(H_2) = \theta(H'_2)$, $cons(H_3) = \theta(H'_3)$, and $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$.

Then $\alpha(\sigma_i) = \langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T \rangle_n$. We show that the ω_t **Apply** transition is applicable to $\alpha(\sigma_i)$, namely

- There exists a (renamed apart) rule in P of the form

  $$r'' \ @ \ H''_1 \setminus H''_2 \iff g'' \mid C''$$

  This is satisfied by $r = r''$, $H''_1 = (H'_1, d_j, H'_2)$, $H''_2 = H'_3$, $g'' = g$ and $C'' = C$. In other words, the same rule as above.

- There exists a matching substitution θ'' such that $cons(H_1, c, H_2) = \theta''(H''_1)$ and $cons(H_3) = \theta''(H''_2)$. This is satisfied by $\theta'' = \theta$, the same matching substitution from above.

- The guard g'' is satisfied, i.e. $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_{r''}(\theta'' \wedge g'')$. This is satisfied because $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g)$ holds, and $r'' = r$, $\theta'' = \theta$ and $g'' = g$.

- The tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$. This is directly satisfied from above.

Hence

$$\begin{array}{l}\langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T \rangle_n \rightarrowtail_{apply} \\ \qquad \langle \theta(C) \uplus no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, B, T' \rangle_n\end{array}$$

Now $\langle \theta(C) \uplus no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, B, T' \rangle_n = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{apply} \alpha(\sigma_{i+1})$ where the ω_t transition is **Apply**. Hence Case 2 above is satisfied.
CASE **Default**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i : j|A], S, B, T \rangle_n \rightarrowtail_{default} \langle [c\#i : j+1|A], S, B, T \rangle_n$$

Then $\alpha(\sigma_i) = \alpha(\sigma_{i+1}) = \langle no\_id(A), S, B, T \rangle_n$. Hence Case 1 above is satisfied.

Therefore, for all ω_r derivations D, $\alpha(D)$ is a ω_t derivation. □

We have shown that every ω_r derivation has a corresponding ω_t derivation given by function α. But in order to show the correspondence of the semantics we also need to show that the terminating ω_r derivations map to terminating ω_t derivations. First we need to define what subset of all ω_r execution states need to be considered.

Definition 16 (Reachability) *An execution state σ (of either semantics) is* reachable *if there exists an initial state $\sigma_0 = \langle G, \emptyset, true, \emptyset \rangle_1$ such that there exists a derivation $\sigma_0 \rightarrowtail^* \sigma$.* □

Not all states are reachable.

Example 24 *The following execution state is not reachable with respect to the* `gcd` *program from Example 14.*

$$\langle [], \{gcd(0)\#1\} \rangle_2$$

If a `gcd(0)` *constraint is present in the store, then at some stage in the derivation that constraint must have been active. When it was active the first rule must have fired, hence the constraint must have been deleted. Therefore the above program state is not reachable.* □

Reachability is generally undecidable for programming languages and this certainly applies to CHRs.

Before we state the main theorem we need to prove two lemmas related to reachability of ω_r states, i.e. given a state of a specified form, together with some additional assumptions, can we find a future state in the derivation that matches some other specified form.

The first lemma says that if we have some prefix of constraints on the top of the execution stack, then eventually those constraints will be removed assuming termination and non-failure.

Lemma 1 (Intermediate States 1) *Let D be a finite ω_r derivation from an execution state σ of the form $\langle A_p +\!\!+ A_s, S, B, T \rangle_n$ (for non-empty A_s) to some non-false final state. Then there exists an intermediate state σ_k of the form $\langle A_s, S_k, B_k, T_k \rangle_{n_k}$ (the same A_s) in D.*

Proof. Direct proof. We define an abstraction function β which takes a suffix A'_s and a state $\langle A'_p +\!\!+ A'_s, S', B', T'\rangle_{n'}$ where the execution stack ends with the suffix A'_s, and returns the length of the prefix A'_p.

$$\beta(A'_s, \langle A'_p +\!\!+ A'_s, S', B', T'\rangle_{n'}) = len(A'_p)$$

Where function *len* returns the length of a sequence defined in the standard way. The function $\beta(A'_s, \sigma')$ is undefined if the execution stack of σ' does not end with A'_s.

The function $\beta(A_s, \sigma)$ is well-defined for suffix A_s and initial state σ from above. Let σ_f be the final state in derivation D. Since σ_f is non-$false$, it must be of the form $\langle [], S_f, B_f, T_f\rangle_{n_f}$. As A_s is non-empty by assumption, function $\beta(A_s, \sigma_f)$ is undefined for σ_f. Therefore there must exist two consecutive states σ_i and σ_{i+1} in D such that $\beta(A_s, \sigma_i)$ is defined but $\beta(A_s, \sigma_{i+1})$ is not.

We consider the possible transitions between σ_i and σ_{i+1}.
CASE **Activate**, **Reactivate** and **Default**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i)$ is also defined. Hence we can exclude these cases.
CASE **Drop**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) - 1$ is defined if $\beta(A_s, \sigma_i) \geq 1$, otherwise $\beta(A_s, \sigma_{i+1})$ is undefined.
CASE **Solve**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) + len(S_1) - 1$ (where S_1 represents that constraints woken up by the wakeup policy) is defined if $len(S_1) \geq 1$ or $\beta(A_s, \sigma_i) \geq 1$, otherwise $\beta(A_s, \sigma_{i+1})$ is undefined.
CASE **Simplify**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) + len(C) - 1$ (where C is the body of the rule that fired) is always defined since $len(C) \geq 1$ (the body of a rule must be at least of length 1). Hence we can exclude this case.
CASE **Propagate**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) + len(C)$ (where C is the body of the rule that fired) is always defined. Hence we can exclude this case.

Hence the only possible transitions between σ_i and σ_{i+1} are

1. **Drop** when $\beta(A_s, \sigma_i) = 0$; and
2. **Solve** when $\beta(A_s, \sigma_i) = 0$ and $len(S_1) = 0$.

In either case, $\beta(A_s, \sigma_i) = 0$ holds iff σ_i is of the form $\langle A_s, S_i, B_i, T_i\rangle_{n_i}$, hence $\sigma_k = \sigma_i$ directly satisfies our hypothesis. □

The second lemma says that if we have a newly activated constraint (with occurrence number 1), and some occurrence k in the program for that constraint, then the active constraint will eventually reach occurrence k assuming termination, non-failure and the active constraint is never deleted.

Lemma 2 (Intermediate States 2) *Let $\sigma = \langle [c\#i : 0|A], \{c\#i\} \uplus S, B, T\rangle_n$ be an ω_r execution state and D a derivation from σ to some non-$false$ final state σ_f such that $c\#i \in S_f$. Then for all programs P and occurrences k of predicate c in P, there exists an intermediate state σ_k of the form $\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_{n_k}$ in D.*

Proof. By induction.
Base case: $k = 1$. Then $\sigma_k = \sigma$ is of the appropriate form.
Induction Step: Assume that for all programs P and occurrences k of predicate c in P, there exists an intermediate state σ_k of the form $\sigma_k = \langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n$. We show how to find state $\sigma_{k+1} = \langle [c\#i : k+1|A], \{c\#i\} \uplus S_{k+1}, B_{k+1}, T_{k+1}\rangle_{n_{k+1}}$.

Note that because derivation D is finite, there can only be a finite number of states of the form σ_k, so w.l.o.g assume σ_k is the last state in D of the required form.

We consider all possible ω_l transitions applicable to σ_k.
CASE **Solve**, **Activate** and **Reactivate**: None of these are applicable to a state of the form σ_k;
CASE **Drop**: Not applicable since this means occurrence k does not exist, which violates the induction hypothesis;
CASE **Simplify**: This will delete the active constraint which violates our assumption that $c\#i$ is never deleted throughout derivation D.
CASE **Propagate**: Then

$$\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n \rightarrowtail_{propagate} \langle C +\!\!+ [c\#i : k|A], \{c\#i\} \uplus S'_k, B'_k, T'_k\rangle_n$$

We can now apply Lemma 1 to find a future state σ'_k in derivation D of the form

$$\langle [c\#i : k|A], \{c\#i\} \uplus S''_k, B''_k, T''_k\rangle_{n''}$$

But σ'_k is in the same form as σ_k, which contradicts our assumption that σ_k was the last state in the derivation of that form.

Thus the only transitions applicable to σ_k is **Default**.

$$\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n \rightarrowtail_{default} \langle [c\#i : k+1|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n$$

The new state is of the required form for $k + 1$.

Therefore for all programs P and occurrences k of predicate c in P, there exists an intermediate state σ_k of the form $\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_{n_k}$ in D, provided D starts with state a state of the form $\langle [c\#i : 1|A], \{c\#i\} \uplus S, B, T\rangle_n$, and does not delete $c\#i$. □

We can now show that reachable final ω_r states map to reachable[8] final states under ω_t.

Theorem 3 (Final States) *Let σ be a reachable final state under ω_r, then $\alpha(\sigma)$ is a reachable final state under ω_t.*

Proof. By contradiction. Assume that $\alpha(\sigma)$ is not a final state, i.e. $\alpha(\sigma)$ can fire at least one ω_t transition.

[8]Reachability is obviously preserved thanks to Theorem 2

Because σ is a *reachable* final state, these exists an initial state σ_0 such that $\sigma_0 \rightarrowtail_D^* \sigma$ for some ω_r derivation D. For convenience, we rename $\sigma = \sigma_f$ and name every state in the derivation $\sigma_i = \langle A_i, S_i, B_i, T_i \rangle_{n_i}$ for some i such that $D = \sigma_0 \rightarrowtail \sigma_1 \rightarrowtail ... \rightarrowtail \sigma_f$. By Theorem 2, $\alpha(D)$ is a ω_t derivation from $\alpha(\sigma_0)$ to $\alpha(\sigma_f)$.

Execution state σ_n is a final state, therefore $\sigma_n = \langle [], S_f, B_f, T_f \rangle_{m_f}$ (i.e. the execution stack A_f is empty). The other possibility is $\sigma_n = false$, but then $\alpha(\sigma_n) = false$ which is also a final state. Hence $\alpha(\sigma_n) = \langle \emptyset, S_f, B_f, T_f \rangle_{m_f}$, and only the **Apply** transition is applicable to such a ω_t state (the goal field is empty).

Let $H_1 \uplus H_2 \subseteq S_f$, and let $(r \ @ \ H_1' \setminus H_2' \iff g \mid C)$ be the instance of the rule that matches with substitution θ satisfying $cons(H_1) = \theta(H_1')$ and $cons(H_2) = \theta(H_2')$. We also know that $\mathcal{D} \models_{\mathcal{S}} B_f \rightarrow \exists_r(\theta \wedge g)$, and $id(H_1) +\!\!+ id(H_2) +\!\!+ [r] \notin T_f$ otherwise **Apply** is not applicable.

Consider the derivation D. The built-in store is monotonically increasing throughout the derivation, i.e. for all built-in stores B_i and B_j from states $\sigma_i, \sigma_j \in D$, if $i < j$ then $B_i \subseteq B_j$ (by treating B_i and B_j as multisets). This is easily verified by observing none of the ω_r derivations delete constraints from the built-in store. If for some σ_i in D we have that $\mathcal{D} \models_{\mathcal{S}} B_i \rightarrow \exists_r(\theta \wedge g)$, then for all $j > i$ we have that $\mathcal{D} \models_{\mathcal{S}} B_j \rightarrow \exists_r(\theta \wedge g)$ by the monotonicity requirement for $(\mathcal{D} \models_{\mathcal{S}})$.

For final state σ_f, the guard holds, i.e. $\mathcal{D} \models_{\mathcal{S}} B_f \rightarrow \exists_r(\theta \wedge g)$. Therefore there must exist a *first* state $\sigma_b \in D$ such that the guard holds, i.e. $\mathcal{D} \models_{\mathcal{S}} B_b \rightarrow \exists_r(\theta \wedge g)$, but not for σ_j where $j < b$, i.e. $\mathcal{D} \not\models_{\mathcal{S}} B_j \rightarrow \exists_r(\theta \wedge g)$. Note the index '$b$' in σ_b stands for "*b*uilt-in" – the first state where the built-in store entails the guard g.

Let σ_s be the *first* execution state in D where $H_1 \uplus H_2 \subseteq S_s$ where S_s is the CHR store of σ_s. Such a state must exist because $H_1 \uplus H_2 \subseteq S_f$, where S_f is the CHR store of final state σ_f. Here the 's' stands for "CHR *s*tore" – the first state where all constraints $H_1 \uplus H_2$ are in the CHR store.

We know that in derivation D, the states σ_b and σ_s must be present. There are two cases we need to consider, namely $b \leq s$ and $b > s$.
CASE $b \leq s$: Let $c\#i \in H_1 \uplus H_2$ be the CHR constraint such that $c\#i \in S_s$ (the store for σ_s) but $c\#i \notin S_{s-1}$ (the store for σ_{s-1}), i.e. $c\#i$ is the last constraint in $H_1 \uplus H_2$ to be added to the store. The ω_r transition between σ_{s-1} and σ_s must be **Activate**, since this is the only transition that will add a CHR constraint into the store. **Activate** also activates the topmost constraint, so for some A_s' and S_s' we have $\sigma_s = \langle [c\#i : 1 | A_s'], \{c\#i\} \uplus S_s', B_s, T_s \rangle_{n_s}$.

Let k be the occurrence of predicate c in rule r (from above) that matched with c when we applied the **Apply** transition to $\alpha(S_f)$. By Lemma 2 there exists at least one future state σ_t of the form $\langle [c\#i : k | A_t'], \{c\#i\} \uplus S_t', B_t, T_t \rangle_{n_t}$. Since D is finite, there must be a *last* state in the form σ_t, so w.l.o.g. assume that σ_t is the last state in D of the above form. The only possible applicable transition that doesn't violate one of our assumptions is **Propagate**, because:

1. **Solve**, **Activate**, **Reactivate** and **Drop** are directly not applicable to a state in the form of σ_t;

2. **Default** is not applicable, since we know this state could potentially fire the **Propagate** transition on rule r because $H_1 \uplus H_2 \subseteq S_t$ matches against the head of rule r with matching substitution θ (where θ is exactly the same as the one from above). We know that $\mathcal{D} \models_{\mathcal{S}} B_t \rightarrow \exists_r(\theta \wedge g)$ because of the assumption $b \leq s$ and $id(H_1) +\!\!+ id(H_2) +\!\!+ [r] \notin T_t$ because no such entry appears in the final state;

3. **Simplify** is not applicable since it will delete the active $c\#i$ constraint, violating our assumption that $c\#i$ appears in the final store of σ_f.

So the transition applied to σ_t in D must be **Propagate** (on a different matching).

$$\langle [c\#i : k|A'_t], \{c\#i\} \uplus S'_t, B_t, T_t\rangle_{n_t} \rightarrowtail_{propagate} \langle C +\!\!+ [c\#i : k|A'_t], \{c\#i\} \uplus S'_t, B'_t, T'_t\rangle_{n_t}$$

We can now apply Lemma 1 to find a future state σ_u in derivation D of the form $\langle [c\#i : k|A'_t], \{c\#i\} \uplus S'_u, B_u, T_u\rangle_{n_u}$. But σ_u is in the same form as σ_t which contradicts our assumption that σ_t was the last such state.
CASE $b > s$:

Consider the state σ_{b-1}, i.e. the state just before the built-in store satisfies the guard g.

The only transition that modifies the built-in constraint store is **Solve**, hence this must be the transition between σ_{b-1} and σ_b. By the *lower bound* condition of a wakeup policy used by **Solve**, there must be a constraint $c\#i \in H_1 \uplus H_2$ such that $c\#i$ is an element of the constraints woken up by the wakeup policy. Therefore σ_b must be of the form $\langle A_p +\!\!+ [c\#i|A_s], \{c\#i\} \uplus S'_b, B_b, T_b\rangle_{n_b}$. In other words, after applying **Solve** on σ_{b-1}, the constraint $c\#i$ must appear somewhere on the execution stack.

We can now apply Lemma 1 to derive a future state σ'_b of the form $\langle [c\#i|A_s], \{c\#i\} \uplus S''_b, B'_b, T'_b\rangle_{n'_b}$. The only transition applicable to such a state is **Reactivate**, hence

$$\langle [c\#i|A_s], \{c\#i\} \uplus S''_b, B'_b, T'_b\rangle_{n'_b} \rightarrowtail_{reactivate} \langle [c\#i : 1|A_s], \{c\#i\} \uplus S''_b, B'_b, T'_b\rangle_{n'_b}$$

Now this new state is in the same form as σ_s from the $b \leq s$ case (see above), hence we can apply the same argument as before to derive the same contradiction. □

Theorem 2 and Theorem 3 show that the refined operational semantics correctly implement the theoretical operational semantics. Hence, the soundness and completeness results for CHRs under the theoretical operational semantics hold under the refined operational semantics ω_r.

First we define the logical interpretation of execution states for the refined operational semantics.

Definition 17 (Logical Interpretation of Refined Execution States) *The logical interpretation of an ω_r execution state σ is*

$$[\![\sigma]\!] = [\![\alpha(\sigma)]\!]$$

□

Now we can (re)state the soundness and completeness results for the refined operational semantics (the results for the theoretical semantics is in [32]).

Corollary 1 (Soundness) *Let P be a CHR program, σ_0 be an initial state and σ_f be a state such that $\sigma_0 \rightarrowtail^* \sigma_f$ under the ω_r semantics, then $([\![P]\!] \wedge \mathcal{D}) \models \forall([\![\sigma_0]\!] \leftrightarrow [\![\sigma_f]\!])$.*

Proof. Directly from Theorem 2, the soundness result for CHRs [32] and the identity $[\![\alpha(\sigma)]\!] = [\![\sigma]\!]$. □

Corollary 2 (Completeness) *Let P be a CHR program, σ_C and σ_0 be initial states, σ_f be a state such that $\sigma_0 \rightarrowtail^* \sigma_f$ under the ω_r semantics and $([\![P]\!] \wedge \mathcal{D} \models \forall(\sigma_C \leftrightarrow [\![\sigma_0]\!])$, then $([\![P]\!] \wedge \mathcal{D} \models \forall(\sigma_C \leftrightarrow [\![\sigma_f]\!])$.*

Proof. Directly from Theorem 2, the soundness result for CHRs [32] and the identity $[\![\alpha(\sigma)]\!] = [\![\sigma]\!]$. □

3.5.1 Termination

Termination of CHR programs is obviously a desirable property. Thanks to Theorems 2 and 3, termination of ω_t programs ensures termination of ω_r.

Firstly we need to show that all ω_r derivations consisting only of **Reactivate**, **Drop** and **Default** transitions are finite. Notice that these are the transitions that disappear after function α has been applied to a ω_r derivation (see the proof of Theorem 2).

Lemma 3 *Let σ be an ω_r execution state, then there is no infinite ω_r derivation $\sigma \rightarrowtail^\infty$ consisting of only **Reactivate**, **Drop** and **Default** transitions.*

Proof. By constructing a well founded order over such derivations. Firstly define a ranking (abstraction) function *rank* that maps ω_r execution states and a CHR program P to a triple of non-negative integers.

$$rank(\langle A, S, B, T\rangle_i, P) = (len(A), len(nums(A)), total(P) - occ(A))$$

Where function *len* maps a sequence to the length of that sequence, defined in the standard way. Function *nums* maps a sequence of constraints to a sequence of numbered constraints by filtering out all non-numbered and active constraints.

$$\begin{array}{ll} nums([]) & = [] \\ nums([c|A]) & = nums(A) \\ nums([c\#i|A]) & = [c\#i] +\!\!+ nums(A) \\ nums([c\#i : j|A]) & = nums(A) \end{array}$$

Function *occ* maps a sequence of constraints A to the occurrence number of the top-most active constraint if it exists; or 0 otherwise.

$$\begin{array}{ll} occ([]) & = 0 \\ occ([c|A]) & = 0 \\ occ([c\#i|A]) & = 0 \\ occ([c\#i:j|A]) & = j \end{array}$$

Finally function *total* maps a program P to the total number of constraints in the heads of every rule plus one.

$$\begin{array}{ll} total([]) & = 1 \\ total([(r \ @ \ H_1 \setminus H_2 \iff g \mid C)|P]) & = len(H_1) + len(H_2) + total(P) \end{array}$$

Let $\prec$ be the standard lexicographical tuple ordering. For all reachable execution states σ and programs P, $rank(\sigma, P) \succeq (0, 0, 0)$. This directly follows from the fact that $len(A) \geq 0$ for all sequences A, and $total(P) \geq occ(A)$ for all sequences of constraints A and programs P. Thus ordering over the ranks of execution states is *well-founded*, i.e. no infinite decreasing chains of execution state rankings $rank(\sigma_0, P) \succ rank(\sigma_1, P) \succ ....$

Next we show that for all execution states σ and σ' such that $\sigma \rightarrowtail \sigma'$ by transition **Reactivate**, **Drop** or **Default**, then $rank(\sigma', P) \prec rank(\sigma, P)$.
CASE **Drop**: $\sigma \rightarrowtail \sigma'$ is of the form

$$\langle [c\#i:j|A], S, B, T\rangle_n \rightarrowtail_{drop} \langle A, S, B, T\rangle_n$$

If $rank(\sigma, P) = (x_1, x_2, x_3)$, then $rank(\sigma', P) = (x_1 - 1, x_2, x_3')$ for some x_3'. Hence $rank(\sigma', P) \prec rank(\sigma, P)$.
CASE **Reactivate**: $\sigma \rightarrowtail \sigma'$ is of the form

$$\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail_{reactivate} \langle [c\#i:1|A], S, B, T\rangle_n$$

If $rank(\sigma, P) = (x_1, x_2, x_3)$, then $rank(\sigma', P) = (x_1, x_2 - 1, x_3')$ for some x_3'. Hence $rank(\sigma', P) \prec rank(\sigma, P)$.
CASE **Default**: $\sigma \rightarrowtail \sigma'$ is of the form

$$\langle [c\#i:j|A], S, B, T\rangle_n \rightarrowtail_{default} \langle [c\#i:j+1|A], S, B, T\rangle_n$$

If $rank(\sigma, P) = (x_1, x_2, x_3)$, then $rank(\sigma', P) = (x_1, x_2, x_3 - 1)$. Hence $rank(\sigma', P) \prec rank(\sigma, P)$.

Thus we have established a termination order, thus proving derivations consisting of only **Reactivate**, **Drop** and **Default** must be finite. □

We can now state the main termination result.

Lemma 4 *Let σ_0 be an ω_r execution state. If every derivation for $\alpha(\sigma_0)$ terminates under ω_t, then every derivation for σ_0 also terminates under ω_r.*

Proof. By contradiction. Assume $\alpha(\sigma_0)$ terminates under ω_t, but not for σ_0 with respect to ω_r. Then there exists an infinite ω_r derivation D starting from σ_0. By Theorem 2 there must be a corresponding derivation $\alpha(D)$ from initial state $\alpha(\sigma_0)$ with respect to ω_t. By assumption, $\alpha(D)$ must be finite.

We partition derivation D into infinitely many subderivations $D_0 +\!\!+ D_1 +\!\!+ D_2 +\!\!+ ...$ as follows. Each D_i is a finite sub-derivation of D starting from the last state in D_{i-1} for $i > 0$ (or σ_0 otherwise) to a state σ_i such that the transition between the state proceeding σ_i and σ_i in D is either **Solve**, **Activate**, **Simplify** or **Propagate**. All other transitions in D_i must be **Reactivate**, **Drop** and **Default**. In other words, $D_i = D'_i \rightarrowtail_i \sigma_i$ where D'_i is a (possibly trivial) subderivation of D consisting only of **Reactivate**, **Drop** and **Default** transitions, and transition $\rightarrowtail_i$ to state σ_i is either **Solve**, **Activate**, **Simplify** or **Propagate**.

Note that it is always possible to partition D in this way. Otherwise suppose that $D = D_0 +\!\!+ D_1 +\!\!+ D_2 +\!\!+ ... +\!\!+ D_n +\!\!+ D'$ where it is not possible to further partition D', then D' must not contain a **Solve**, **Activate**, **Simplify** or **Propagate** transition. Then Lemma 3 implies D' is finite, thus D is finite, which directly contradicts our initial assumption that D is infinite. So $D = D_0 +\!\!+ D_1 +\!\!+ D_2 +\!\!+ ...$ for infinitely many D_i.

Now $\alpha(D) = \alpha(D_0 +\!\!+ D_1 +\!\!+ D_2 +\!\!+ ...) = \alpha(D_0) +\!\!+ \alpha(D_1) +\!\!+ \alpha(D_2) +\!\!+ ....$ Each D_i contains one **Solve**, **Activate**, **Simplify** or **Propagate** transition, hence each $\alpha(D_i)$ has non-zero length (see the proof of Theorem 2). As the length of $\alpha(D)$ is the sum of the (non-zero) lengths of every $\alpha(D_i)$, and there are infinitely many $\alpha(D_i)$, then $\alpha(D)$ has infinite length which is a contradiction. □

The converse is clearly not true, as shown in Example 14.

In practice, proving termination for CHR programs under the theoretical operational semantics is quite difficult (see [33] for examples and discussion). It is somewhat simpler for the refined operational semantics but, just as with other programming languages, this is simply left to the programmer.

3.5.2 Confluence

Both operational semantics for CHRs are nondeterministic, therefore the property of *confluence* (which guarantees the same result no matter the order transitions are applied) is essential from a programmer's point of view. Without it the programmer cannot anticipate the answer that will arise from a goal.

There are two main sources of nondeterminism from the refined semantics. The first is from the **Solve** transition, where the order (and number of repeats) of the woken up constraints are (re)added to the execution stack is unspecified. The second is from the **Simplify** and **Propagate** transitions, where there may be more than one choice for choosing matching partner constraints.

Example 25 *Consider the* `database` *program from Example 15. The occurrences for* `lookup` *have been labelled.*

```
l1 @ entry(Key,Val) \ lookup(Key,ValOut)₁ <=> ValOut = Val.
l2 @ lookup(_,_)₂ <=> fail.
```

Consider the following (simplified) execution state with two database entries for the same key, and an active ***lookup*** *constraint on the stack.*

$$\langle [\texttt{lookup}(key, V)\#3 : 1], \{\texttt{lookup}(key, V)\#3, \\ \texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, true\rangle_4$$

Now the active ***lookup*** *constraint is at occurrence* 1 *from rule* ***l1*** *above, so* ***Simplify*** *is applicable matching against either* $\texttt{entry}(key, cat)$ *or* $\texttt{entry}(key, dog)$ *from the store. Depending on what matching is chosen (both are equally valid), the resulting state after* ***Simplify*** *is*

$$\langle [], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, V = cat\rangle_4$$

or

$$\langle [], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, V = dog\rangle_4$$

Since these are both final states are not the same result, it follows that the ***database*** *program is non-confluent.* □

In order to properly formalise the property of confluence first we need to define what it means to get the "same result". Unfortunately, a straightforward syntactic comparison is too strong in general, since we do not care about constraint numbering and similar things. We do however care about propagation histories, because "equivalent" states should be similarly applicable to the same set of rules. We define a mapping which extracts the part of a propagation history that we care about as follows.

Definition 18 (Live History) *Function alive is a bijective mapping from a CHR store* S *and a propagation history to a propagation history defined as follows.*

$$\begin{array}{lll} alive(S, \emptyset) & = \emptyset & \\ alive(S, \{t\} \uplus T) & = alive(S, t) \uplus alive(S, T) & \\ alive(S, t +\!\!+ [\_]) & = \emptyset & \text{if } \exists i \in t \text{ such that } \forall c(c\#i \notin S) \\ alive(\_, t) & = \{t\} & \text{otherwise} \end{array}$$

□

In other words, $alive(S, T)$ is propagation history T where all entries with numbers for deleted (i.e. not alive) constraints have been removed. Interestingly, $alive(S, T)$ can only have entries on propagation rules (otherwise one of the numbers in the entry must be dead).

We can now formally define variance between two states.

Definition 19 (Variants) *Two states*

$$\sigma_1 = \langle A_1, S_1, B_1, T_1\rangle^{\mathcal{V}}_{i_1} \qquad \text{and} \qquad \sigma_2 = \langle A_2, S_2, B_2, T_2\rangle^{\mathcal{V}}_{i_2}$$

(from either semantics) are variants *if there exists a renaming* ρ *on variables not in* $\mathcal{V}$ *and a mapping* ϱ *on constraint numbers such that*

1. $\rho \circ \varrho(A_1) = A_2$ *(sequence equality for* ω_r*, multiset equality for* ω_t*);*
2. $\rho \circ \varrho(S_1) = S_2$*;*
3. $\mathcal{D} \models_{\mathcal{S}} (\bar{\exists}_{\mathcal{V}} \rho(B_1) \leftrightarrow \bar{\exists}_{\mathcal{V}} B_2)$*; and*
4. $\varrho \circ alive(S_1, T_1) = alive(S_2, T_2)$*.*

Otherwise the two states are variants if $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset} B_1$ *and* $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset} B_2$ *(i.e. both states are* $false$*).* □

In other words, we consider two (non-$false$) states σ_1 and σ_2 to be variants (i.e. the "same result") if the two goals (or execution stacks) and CHR stores are the same, the built-in solver can prove that the built-in stores are logically equivalent, and the "live" parts of the propagation histories are the same, all modulo variables not appearing in $\mathcal{V}$ and constraint numbering.

We can now define *joinability*, which is the property that two execution states reduce to the same answer.

Definition 20 (Joinable) *Two states* σ_1 *and* σ_2 *are* joinable *if there exists states* σ_1' *and* σ_2' *such that* $\sigma_1 \rightarrowtail^* \sigma_1'$ *and* $\sigma_2 \rightarrowtail^* \sigma_2'$ *and* σ_1' *and* σ_2' *are variants.* □

Now we can formally define confluence as follows.

Definition 21 (Confluence) *A CHR program* P *is* confluent *with respect to operational semantics* ω *if the following holds for all states* σ_0*,* σ_1 *and* σ_2 *where* σ_0 *is a reachable state: If* $\sigma_0 \rightarrowtail^*_\omega \sigma_1$ *and* $\sigma_0 \rightarrowtail^*_\omega \sigma_2$ *then* σ_1 *and* σ_2 *are joinable with respect to* σ_0*.* □

This definition is slightly stronger than the classical definition in [1] since we require σ_0 to be a *reachable* state. This is important, since the programmer generally only cares about reachable states. Our definition is stronger, so confluence under the classical definition implies confluence under our new definition.

Example 26 *The* `gcd` *program from Example 14 is confluent under both operational semantics (although it may not terminate under the theoretical semantics). This is because any final state derived from the initial goal* $\texttt{gcd}(i_1), ..., \texttt{gcd}(i_n)$ *must contain only the constraint* $\texttt{gcd}(gcd(i_1, ..., i_n))$ *in the store. The built-in stores must also be equivalent once all temporary variables (created by matching constraints against rules) are renamed. Hence the* `gcd` *program is confluent.* □

Confluence of the theoretical operational semantics of CHR programs has been extensively studied [32, 1, 4]. Abdennadher [1] provides a decidable confluence test for the theoretical semantics of terminating CHR programs. Essentially, it relies on computing critical pairs where two rules can possibly be used, and showing that each of the two resulting states lead to equivalent states.

Just as with termination, confluence under ω_t implies confluence under ω_r provided the program also terminates under ω_r.

Corollary 3 *If CHR program P is terminating under ω_r, and confluent under ω_t, then it is also confluent under ω_r.*

Proof. By contradiction. Assume that P is confluent under ω_t, but not confluent with respect to ω_r. Then there exists a reachable ω_r state σ_0 such that $\sigma_0 \rightarrowtail^* \sigma_1'$ and $\sigma_0 \rightarrowtail \sigma_2'$ where σ_1' and σ_2' are not joinable. Let σ_1 and σ_2 be final states derived from σ_1' and σ_2' respectively, i.e.

$$\sigma_1' \rightarrowtail^* \sigma_1 = \langle A_1, S_1, B_1, T_1 \rangle_{i_1}^{\mathcal{V}}$$
$$\sigma_2' \rightarrowtail^* \sigma_2 = \langle A_2, S_2, B_2, T_2 \rangle_{i_2}^{\mathcal{V}}$$

Where $A_1 = A_2 = []$ or both σ_1 and σ_2 are $false$. Note that it always possible to find σ_1 and σ_2 because of the assumption of termination under ω_r. As σ_1' and σ_2' are not joinable both σ_1 and σ_2 cannot be variants. Note also that by construction $\sigma_0 \rightarrowtail^* \sigma_1$ and $\sigma_0 \rightarrowtail^* \sigma_2$.

By Theorem 2, there exists two derivations under ω_t:

$$\alpha(\sigma_0) \rightarrowtail^* \alpha(\sigma_1) = \langle no\_id(A_1), S_1, B_1, T_1 \rangle_{i_1}^{\mathcal{V}}$$
$$\alpha(\sigma_0) \rightarrowtail^* \alpha(\sigma_2) = \langle no\_id(A_2), S_2, B_2, T_2 \rangle_{i_2}^{\mathcal{V}}$$

By assumption P is confluent under the theoretical operational semantics, therefore both $\alpha(\sigma_1)$ and $\alpha(\sigma_1)$ must be joinable. By Theorem 3 both $\alpha(\sigma_1)$ and $\alpha(\sigma_2)$ are final states, therefore to be joinable they must be variants.

There are two cases to consider.
CASE 1: $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset} B_1$ and $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset} B_2$:
(I.e. both states are $false$). Then σ_1 and σ_2 must be variants since the built-in store is unaffected by abstraction function α.
CASE 2: There exists a renaming ρ on variables not in σ_0 and a mapping ϱ on constraint numbers such that $\rho \circ \varrho(S_1) = S_2$, $\mathcal{D} \models_{\mathcal{S}} (\bar{\exists}_{\mathcal{V}} \rho(B_1) \leftrightarrow \bar{\exists}_{\mathcal{V}} B_2)$ and $\varrho \circ alive(S_1, T_1) = alive(S_2, T_2)$. Note that because σ_1 and σ_2 are final states, their execution stacks are empty, i.e. $A_1 = A_2 = []$. Therefore σ_1 and σ_2 are variants by definition.

Both cases directly contradict our assumption that σ_1 and σ_2 are not variants. Therefore if program P which is terminating under ω_r is confluent under ω_t, then it is also confluent under ω_r. □

Example 27 *Both the* `gcd` *and* `leq` *programs (from Example 14 and Example 1) are terminating under ω_r, and are confluent under ω_t. Therefore, by Corollary 3, confluence under ω_r immediately follows.* □

The converse of Corollary 3 is not true, as shown by the following simple example.

Example 28 *The following program is confluent under ω_r (since an active* `p` *constraint always fires rule* `r1`*).*

```
r1 @ p <=> true.
r2 @ p <=> false.
```

However the program is not confluent under ω_t (since `p` *can fire either* `r1` *or* `r2` *resulting in non-joinable states).* □

This shows that the set of confluent programs under ω_r is larger than the same set under ω_t.

Later in this thesis, Chapter 6 is devoted to a practical analysis for confluence under the refined semantics.

3.6 Related Work

The presentation of the theoretical operational semantics of CHRs differs from others that have appeared in past literature, e.g. in [32, 1], in several ways. In this section we argue that our formalisation subsumes the previous versions.

The main difference is the interpretation of simpagation rules. Previous versions of the operational semantics treated simpagation rules as shorthand for simplification rules. Specifically, a simpagation rule of the form

$$h_1, \ldots, h_l \backslash h_{l+1}, \ldots, h_n \iff g \mid b_1, \ldots, b_m$$

is treated as a simplification rule of the form

$$h_1, \ldots, h_l, h_{l+1}, \ldots, h_n \iff g \mid h_1, \ldots, h_l, b_1, \ldots, b_m$$

This translation does not necessarily preserve operational equivalence under (our version of) ω_t, since the copies of $h_1, \ldots, h_l$ will be assigned new constraint numbers when they are (re)executed in the body. This may effect the behaviour of the propagation history.

The new interpretation of simpagation rules effectively extends the operational semantics in [32, 1] (the old interpretation can be emulated under ω_t by translating simpagation rules explicitly). It should be noted that some theoretical results for CHRs, e.g. Abdennadher's confluence test [1], may depend on the old interpretation of simpagation rules.

Another difference between the semantics is the modelling of the built-in solver. Previous versions of the operational semantics assume the built-in solver is complete, whereas we have generalised the semantics to handle incomplete solvers.

Most of the other differences are trivial. For example, we represent the constraint store as a (multi)set of $c\#i$ tuples where each i is unique. In the formalisation presented in [1], the constraint store is represented as a conjunction of constraints, where repeats in the conjunction are allowed (i.e. not idempotent conjunction). Both formalisations of the constraint store are isomorphic.

Propagation histories are also handled differently. In our approach, entries in the propagation history are tuples of constraint numbers (and the rule token), whereas in other versions of the operational semantics (e.g. [1]), entries record the CHR constraints themselves. This is implemented as follows. When a new

constraint c is introduced into the store (via the **Introduce** transition), a so-called *token set*[9] is generated for constraint c. Each token is of the form $r@H'$, where $(r \ @ \ H \implies g \mid B)$ is a propagation rule in P (r is the rule identifier), and H' is a conjunction of constraints in $c \wedge S$ and c is a conjunct in H'. When propagation rule r fires on constraints $H' \subseteq S$, a corresponding token $r@H'$ must be present in the propagation history, and is removed when the rule fires. Notice this is the the opposite approach, removing elements, rather than adding elements to propagation history as the derivation progresses.

Example 29 *For example, consider the following propagation rule.*

```
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Under our representation, firing the propagation rule on the following matching constraints, $\texttt{leq}(A,B)\#1, \texttt{leq}(B,C)\#2$*, adds the entry* $[1,2,\texttt{transitivity}]$ *to the propagation history.*

Under the alternative formalisation, a token $\texttt{transitivity}@(\texttt{leq}(A,B) \wedge \texttt{leq}(B,C))$ *is present in the propagation history and removed when the rule fires.* □

These approaches are isomorphic, that is the operational semantics are equivalent no matter which formalisation of the propagation history is chosen, however our approach more closely mimics what most CHRs systems actually implement.

3.7 Summary

In this chapter we have presented the refined operational semantics of CHRs, which are a popular semantics used by almost all current CHR implementations that we are aware of. The refined operational semantics define a powerful and expressive language, where simple database operations and fixed-point computations are straightforward to implement.

We have proved several important results, such as correctness, soundness, completeness, termination and confluence. The correctness results state that every derivation in the refined semantics map to a derivation under the theoretical semantics with the same answer. Also, every reachable final state in the refined semantics maps to a reachable final state in the theoretical semantics, therefore the refined semantics correctly implement the theoretical semantics. The other results including soundness, completeness, termination and confluence follow from correctness. Soundness and completeness are important results from a theoretical point of view.

Termination is an essential property of any program, including CHR programs. The termination result ensures that a termination proof in the theoretical semantics immediately applies to the refined semantics. This is useful since termination

[9] Although it is called the token "set", it is really a multiset of tokens (repeats are allowed).

of CHRs has been looked at in [33], and therefore these results carry to the refined semantics. It may be easier to prove termination under the refined semantics because of the increased determinism, hence there are less derivations to consider, however it is still left to the programmer.

Although the refined semantics is far more deterministic than the theoretical semantics, some choices such as what order matchings are chosen, and what order non-ground constraints are (re)added to the stack (by **Solve**) are left open. This means that confluence is still important for the programmer to consider. Our confluence result presented in the chapter is of somewhat limited use, but in general we would like a stronger test. Later in Chapter 6 we look at a better confluence analysis for the refined operational semantics of CHRs.

This chapter forms the theoretical basis for the rest of this thesis.

Chapter 4

Basic Compilation

4.1 Introduction

This chapter forms the *practical* basis of this thesis. Here, we explain the *basic compilation* of CHRs into a CLP language such as HAL. We present all of the information required to make a simple "no-frills" CHR compiler that gives reasonable performance on many programs. This will form the basis for more advanced compilation, including optimisation, which is covered later in this thesis.

The specification of the refined operational semantics (see Definition 11) describes a state machine with (reasonably complicated) transitions between states. It is straightforward to implement a naive interpreter for this state machine, and the Chameleon programming language [81] uses such an interpreter. Interpretation is generally slower than compilation, since much of the specification is implemented manually, e.g. the execution stack is a notable example. For compiled CHRs large chucks of the functionality required may already be provided by the target language, e.g. the execution stack becomes the call stack, etc.

Compilation of CHRs is very similar to compiling other programming languages in that it is a multi-phase process: The first phase is parsing and normalisation (desugaring), followed by analysis, optimisation and then finally code generation. Usually analysis and optimisation are optional, and are not covered here. We will briefly look at parsing and program normalisation, but the main focus of this chapter will be on code generation.

Ideally the output of the CHR compiler should be as efficient or better than the code a human would write.

Example 30 *Consider the following program originally from Example 14.*

```
gcd(0) <=> true.
gcd(M) \ gcd(N) <=> M =< N | gcd(M-N).
```

The following is a similar program written in Prolog by a human. It is provided as a benchmark for comparison later on.

```
gcd(N,M,R) :-
     ( M =< N ->
        ( M = 0 ->
           R = N
        ;  gcd(N-M,M,R)
        )
     ;  gcd(M,N,R)
     ).
```

□

Unfortunately the result of the basic compilation of `gcd` will be very different than the human implemented version above, but this shall improve in later chapters.

The rest of this chapter is divided up as follows. First we will briefly look at parsing and program normalisation in section 4.2. Section 4.3 will describe our simple runtime system in preparation for Section 4.4, which describes code generation. Next we devote Section 4.5 to the surprisingly tricky problem of compiling guards. Finally we conclude.

4.2 Parsing and Normalisation

Parsing and normalisation are the first phases in any compilation process. We briefly describe parsing and normalisation of CHR programs.

4.2.1 Parsing

The general problem of parsing any programming language, including CHRs, is well-studied and there are many tools that help automate this process. Since CHRs are usually embedded in a host language (in this case a logic programming language), usually the language's parser is adapted to recognise CHRs. This is the case with all HAL and Prolog implementations of CHRs.

Usually the result of the parsing, assuming no syntax errors, is a list of some representation of the rules in the program. For example, we can represent the rule $(r \ @ \ H_1 \setminus H_2 \iff g \mid B)$ as the term `rule(`r`,`H_1`,`H_2`,`g`,`B`)`. We also translate the head, guard and body into lists of terms, where each term represents a constraint from the original rule. The rule name r is optional, so if it is omitted by the programmer usually the parser will generate one (making sure it is unique with respect to the other rules).

Example 31 *Consider the* `gcd` *program from Example 14. After parsing the program under our scheme the result is the following list of rules.*

```
[ rule(gcd1,[],[gcd(0)],[],[true]),
  rule(gcd2,[gcd(M)],[gcd(N)],[M =< N],[gcd(M-N)])]
```

□

4.2.2 Head and Guard Normalisation

Normalisation is a preprocessing step aimed at making all guards explicit, since non-variable terms and matching variables appearing in the head of a rule or guard actually indicate more guards. *Head normalisation* is achieved by iteratively applying the following steps

1. Rewrite each constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ where c is an n-ary constraint symbol and t_i is a non-variable term, to $c(t_1, \ldots, X, \ldots, t_n)$ and add the constraint $X = t_i$ to the guard.

2. If variable X appears as an argument more than once in the head of a rule, replace one occurrence with a new variable, say X', and add the constraint $X = X'$ to the guard.

After normalisation, each head simply provides the multiset of names of constraints which can match that rule, while the guard indicates which such multisets actually match.

To simplify analysis and compilation, guards are also normalised.

1. Rewrite each constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ where c is an n-ary constraint symbol and t_i is either a non-variable term or a variable which appears as an argument elsewhere in the guard, to $X' = t_i \wedge c(t_1, \ldots, X', \ldots, t_n)$ where X' is a new variable.

2. Rewrite each equation $X = f(t_1, \ldots, t_i, \ldots, t_n)$ where f is an n-ary function and t_i is either a non-variable term or a variable which appears as an argument elsewhere in the guard, to $X' = t_i \wedge X = f(t_1, \ldots, X', \ldots, t_n)$, where X' is a new variable.

3. Add explicit existential quantification for each existentially quantified variable in the guard.

Recall that a variable that appears in the guard but not in the rule head is implicitly existentially quantified.[1] For explicit existential quantification we introduce the notation

`exists [`$X_1, ..., X_n$`] (`g`)`

to indicate that variables $X_1, ..., X_n$ are existentially quantified in guard g. Technically, $X_i \notin vars(C)$ (where C is the rule body) for each $X_i \in X_1, ..., X_n$ in order for X_i to be existentially quantified under Definition 13. If this is not the case, we can use *body normalisation* (see Section 3.4) to rewrite the rule into an equivalent version where the shared variables have been renamed, and the (renamed) guard is (re)executed in the body. For example, the rule

[1] See Definition 13, where existentially quantified variables $\exists \bar{y}$ are precisely the variables in the guard not appearing in the head.

```
elems(Xs) <=> exists [X,Ys] (Xs = [X|Ys]) | e(X), elems(Ys).
```

is equivalent to

```
elems(Xs) <=> exists [X,Ys] (Xs = [X|Ys]) |
                                    Xs = [A|Bs], e(A), elems(Bs).
```

In practice CHR compilers do not perform such transformations, since it means the guard is executed twice. Instead we assume body normalisation has been implicitly applied where appropriate (i.e. we effectively consider the non-normalised rule to be shorthand for the normalised version).

After guard normalisation, each constraint is either an equation of the form $X = Y$, $X = f(Y_1, ..., Y_n)$, or a constraint $c(Y_1, ..., Y_n)$ where $X, Y, Y_1, ..., Y_n$ are all distinct variables. Both head and guard normalisation preserve the operational and declarative meanings of the program. Note that sometimes multiple normalisations are possible, e.g. the head `leq(`X`,`X`)` can be normalised as `leq(`X`,`N`)` or `leq(`N`,`X`)` with guard $X = N$. Such normalisations are always unique up to variable renaming, hence any normalisation can be used.

Example 32 *Consider the following CHR program defining a* ***`length(Xs,L)`*** *constraint which holds if* `L` *is the length of list* ***`Xs`****. Unlike the standard* ***`length`****/2 predicate, this version works in any mode, including when both arguments are fresh variables.*

```
length([],L)     <=> L = 0.
length(Xs,0)     <=> Xs = [].
length([_|Xs],L) <=> L = L1+1, length(Xs,L1).
length(Xs,L)     <=> 1 =< L | Xs = [_|Ys], length(Ys,L-1).
length(_,L)      ==> 0 =< L.
```

The following is the same program after head and guard normalisation has been applied.

```
length(Xs,L)  <=> Xs = []                        | L = 0.
length(Xs,L)  <=> L = 0                          | Xs = [].
length(Xs1,L) <=> exists [A,Xs] (Xs1 = [A|Xs]) |
                          L = L1+1, length(Xs,L1).
length(Xs,L)  <=> exists [L1] (L1 = 0, L > L1) |
                          Xs = [_|Ys], length(Ys,L-1).
length(_,L)   ==> true                           | L >= 0.
```

Thanks to head normalisation all variables appearing in the rule heads are distinct variables. Guard normalisation has made all implicit existential quantification explicit and reduced the guard into the simplified form. □

4.2.3 Program Normalisation

Our representation still closely resembles the original program, but this is not very helpful for the later phases of compilation. Since the refined operational semantics treats a constraint as a call, and checks each occurrence in order, it is useful to create a mapping between the constraint and a list of occurrences for that constraint. This mapping is the normal form of our program.

First we must define a data structure to represent an individual occurrence for a constraint. Suppose we have a rule `rule(`r`,`H_1 ++ $[c]$ ++ H_2`,`H_3`,`g`,`B`)`, then our representation of the occurrence in that rule for c is `occ(`c`,remain,`n`,`H_1 ++ H_2`,`H_3`,`g`,`B`,`r`)`. The first field contains the constraint from the head that must match the active constraint. The second field is the constant `remain`, which represents the fact that this occurrence does not delete the active constraint (in the other case, this field will contain the constant `delete`). The third field n is the occurrence number, which must be calculated during normalisation. For example, if the occurrence of c is the third in the program, then $n = 3$. The fourth field contains the non-deleted part of the head. The rest of the fields, e.g. H_3, g, etc., are exactly the same as in the original rule.

The other case is where the occurrence for c is in the deleted part of the head, i.e. `rule(`r`,`H_1`,`H_2 ++ $[c]$ ++ H_3`,`g`,`B`)`, then the representation of the occurrence is `occ(`c`,delete,`n`,`H_1`,`H_2 ++ H_3`,`g`,`B`,`r`)`.

Example 33 *The list of occurrences for constraint* `gcd`/1 *from Example 31 is the following.*

```
[ occ(gcd(X),delete,1,[],[],[X = 0],[true],gcd1),
  occ(gcd(N),delete,2,[gcd(M)],[],[M =< N],[gcd(M-N)],gcd2),
  occ(gcd(M),remain,3,[],[gcd(N)],[M =< N],[gcd(M-N)],gcd2)]
```

Notice that the head of the first occurrence has been normalised. □

The advantage of the normal form this that all of the information required to compile an individual occurrence is now in one place.

4.3 Runtime Environment

The refined operational semantics defines an *execution state* to be a tuple containing an execution stack, CHR store, built-in store and propagation history. In this section we show how to implement each of these in a pure logic programming language with minimal extensions.

4.3.1 Execution Stack

In the formalisation of the refined operational semantics of CHRs we explicitly represented the execution stack as a sequence of constraints, numbered constraints

and active constraints. In practice the execution stack is nothing more than the ordinary program call stack in HAL or Prolog.

The execution stack starts off as a sequence of (non-numbered) constraints and built-in constraints. Built-in constraints are not treated specially in any way, they are just ordinary HAL procedure calls. CHR constraints are different in the sense that the CHR compiler must generate the required code. Given a CHR constraint in the original CHR program, the CHR compiler generates a predicate, which implements the constraint, with exactly the same interface as the original. The predicate that the compiler generates is called the *top-level predicate* and will be explained in the code generation section.

The execution stack also contains active constraints. An active constraint $p(X_1, ..., X_n)\#I : m$ will be implemented by an *occurrence predicate* with interface `p_`m`(I,`X_1`,...,`X_n`)`. The occurrence predicate will contain the code generated by the compiler for finding matches for the occurrence m of constraint `p`. Occurrence predicates are also *chained*, i.e. `p_`m calls `p_`$(m + 1)$ if the rule cannot fire (i.e. the **Default** transition). Exactly how this is done is left for the code generation section.

Finally, the execution stack contains numbered constraints, which are woken up from the store after a **Solve** transition. In the implementation the **Solve** and **Reactivate** transitions are implemented together, so there is no special representation of numbered constraints on the stack, only active constraints.

4.3.2 CHR Store

CHR constraints in the store are of the form $c\#i$ where i a unique number. A naive implementation of constraint identifiers could use ordinary integers (as in the specification of the refined semantics), however there are reasons why this is an inefficient approach. The operation of testing if a constraint identifier belongs to a deleted constraint turns out to be a useful and common operation. While an ordinary integer has no memory of if it has been deleted or not, a cleverer implementation is to use a fresh Herbrand variable as a constraint identifier. The advantage is that when a constraint is deleted, the variable identifier is bound to some pre-defined atom, e.g. '`deleted`', and then testing if a CHR constraint has been deleted is reduced to testing if the identifier is still a variable (a very fast operation in HAL/Prolog). This approach was first used by the ECL^{i}PS^e CHR compiler [36].

The HAL implementation of constraint identifiers is similar except that instead of using variables a special mutable[2] data structure is used. This data structure has two states: either the constraint identifier belongs to a deleted constraint or not. We refer to an identifier belonging to a deleted constraint as a *dead* identifier, otherwise it is *alive*.

We define the following operations on constraint identifiers.

[2]Changes to this data structure are trailed, so they will be undone on backtracking.

- `new(Id)` – creates a new constraint identifier `Id`;
- `kill(Id)` – marks the constraint identifier `Id` as being dead; and
- `alive(Id)` – succeeds if `Id` is alive, otherwise fails.

In the case of the fresh variable implementation, `new(Id)` is equivalent to a NOP (No OPeration), as this implicitly creates a fresh variable, and `alive(Id)` is equivalent to Prolog's `var(Id)` which succeeds if `Id` is a variable.

A numbered constraint in the CHR store will be represented as the special tuple c `#` i, where c is the constraint and i is the identifier. Our representation of the global CHR store will be a single global list for simplicity. Searching for matching constraints against some rule will involve iterating through this global list. The cost of this simplicity is that searching for matching partners in a list is an $O(N)$ operation, where N is the length of the list.

The CHR store is usually implemented by global variables, which are supported by HAL and some implementations of Prolog.[3] We assume the following abstract operations on the global store.

- `insert(C,Id)` – Insert constraint `C # Id` into the global CHR store;
- `delete(Id)` – Delete the constraint associated with identifier `Id` from the global CHR store and mark `Id` as *dead*; and
- `get_iterator(Ls)` – Binds `Ls` to be a list of all constraints in the global CHR store;

The `get_iterator` operation will be used by the code for finding matchings.

One obvious improvement is to specialise `get_iterator` for each of the predicate symbols of the constraints in the program. For example, if the program defines a constraint `p`/3, then a specialised `get_iterator_p_3` returns an iterator containing only `p`/3 constraints. For simplicity, we will use the non-specialised version for the time being.

4.3.3 Built-in Store

CHR programs may extend zero or more built-in solvers. In all current Prolog implementations of CHRs it is always assumed that there is exactly one built-in solver: the Herbrand equation solver (i.e. ordinary Prolog unification). In general any number of different built-in solvers are possible, but for an implementation, there must be communication between compiled CHR code and these solvers.

Example 34 (Length Constraint) *Consider the* `length` *program from Example 32 which extends both Herbrand and finite domain solvers. Consider the*

[3] Global variables were originally supported by Quintus Prolog [67], but most (if not all) Prologs have something similar.

goal `length(Xs,L), L = 1`. *Its execution will first add the CHR constraint* `length(Xs,L)` *to the store. This constraint cannot by itself cause the application of any of the rules except rule* (5) *which adds the additional constraint* $L \geq 0$. *Next we add the finite domain constraint* $L = 1$ *to the finite domain solver store. This affects rule* (4) *which can now be applied:* `Xs` *is bound to* `[_|Ys]` *and CHR constraint* `length(Xs,L)` *is replaced by* `length(Ys,L1)` *where* $L1 = L - 1 = 0$. *The second rule simplifies* `length(Ys,L1)` *to the Herbrand constraint* `Ys = []`. *Hence, the final solution is* `Xs = [_], L = 1`. □

We can see three kinds of interaction between the CHR solver and the built-in solvers in the example above.

1. The CHR solver adds new constraints to the built-in solvers.

2. The CHR solver asks the built-in solvers whether constraints are *entailed*. This is for testing the guard holds in the **Simplify** and **Propagate** transitions.

3. The built-in solvers must alert the CHR solver whenever non-trivial changes to the built-in store occur. This is to correctly implement the **Solve** transition, which requires all non-ground CHR constraints to be woken up.

Constraint solvers, by definition, provide methods that allow new constraints to be added to their store. The second kind of interaction that needs a well defined interface if we wish to extend an arbitrary built-in solver. We will defer consideration of this until Section 4.5.

For the third type of interaction we use a very simple form of dynamic scheduling. We assume the existence of a special predicate `delay`($Term$`,`Id`,`$Goal$`)`, which delays *Goal* on the condition that any variable in $Term$ has *changed* provided Id, which is a constraint identifier, is still alive. This is very similar to the `delay`/2 predicate introduced in Section 2.3.3, except the first argument may be any term (as opposed to a list of variables) and the Id argument. Thus, with the appropriate delayed goals set, constraints from the CHR store are in effect re-added to the execution stack each time a constraint is added to the built-in store.

It is usually the solver writer's responsibility to implement the `delay` predicate, because the implementation requires intimate knowledge of the inner workings of that solver. If there are multiple solvers then we assume that `delay`/3 is overloaded. Advanced implementations of `delay` are discussed later in Chapter 8. In Prolog the `delay` predicate can be implemented in terms of existing dynamic scheduling constructs, e.g. with attributed variables [44].

In Chapter 8 we also show that the set of constraints woken up by `delay`/3 satisfies the conditions for a *wakeup policy* (see Definition 10), as required by the refined semantics.

4.3.4 Propagation History

The propagation history is very easy to implement naively, but quite challenging to implement efficiently. A naive implementation uses some efficient queryable data structure (e.g. balanced tree or hash table) over the entries. The advantage is that testing if an entry is in the propagation history is very fast, however as program execution proceeds, the propagation history grows in size. Ideally, whenever a constraint is deleted, any entry in the propagation history which contains the corresponding number should also be deleted. We need a mechanism for determining all entries that are associated with a given identifier. In our naive implementation above, there is no better solution than to search through the entire history, which is a relatively expensive $O(N)$ operation (where N is the size of the propagation history).

One solution is to maintain two data structures. One is the propagation history itself, and the other is a mapping from constraint identifiers to entries in the history. The problem with this approach is that is uses more memory.

A better solution is to do amortised update. This works as follows: after some n deletions, the propagation history is purged of all entries containing dead identifiers. Although this is a $O(N)$ operation the number n is configurable: small n means the history will have fewer dead entries but at a higher cost of updating the history more often, whereas large n is the opposite (a classic space versus time tradeoff). Those keen on optimal performance should rely on experimental results in determining the best value for n.

Another approach is to delete entries lazily. For example, if we are searching for an entry in the history and happen to come across a dead entry, then we could delete that entry in-place. This approach is more difficult to implement, and it is possible that dead entries are never deleted (i.e. we may never perform a search that happens to find the dead entry). The current HAL CHR compiler uses this approach.

We will assume a global propagation history, and all interactions with it go through one operation: `check_history(`*Entry*`)` which fails if *Entry* (which is a list of constraint identifiers and the rule name) is already in the global propagation history, or adds it otherwise. In other words, for a unique new entry *Entry*, the first call to `check_history(`*Entry*`)` will succeed, but all subsequent calls with the same *Entry* will fail. Note that for simplicity, the rule name is encoded as a constraint identifier.

4.4 Code Generation

Code generation is the final stage in any compilation process. In this section we give pseudo code and describe what exactly needs to be generated. The only exception is the implementation of the guard test, which is covered by the next section.

```
:- pred p(t_1,...,t_n).                                    (1)
:- mode p(m_1,...,m_n) is d.                               (2)
p(X_1,...,X_n) :-                                          (3)
    new(Id),                                               (4)
    C = p(X_1,...,X_n),                                    (5)
    insert(C,Id),                                          (6)
    delay(C,Id,p_1(Id,X_1,...,X_n)),                       (7)
    p_1(Id,X_1,...,X_n).                                   (8)
```

Figure 4.1: Pseudo code for a top-level predicate.

4.4.1 Top-level Predicate

The top-level predicate is called in place of the original CHR constraint after compilation. Its role is to perform the necessary initialisation, i.e. allocating a new constraint identifier, insertion into the CHR store and setting up appropriate delayed goals on any free variables. Naturally the top-level predicate has the same interface as the original CHR constraint.

For CHR constraint of functor/arity p/n, Figure 4.1 shows the corresponding top-level predicate the CHR compiler generates. Lines (1)-(2) show the `pred` and `mode` declarations. Here $t_1, ..., t_n$, $m_1, ..., m_n$ and d are exactly the argument types, argument modes and determinism for the original CHR constraint p/n. The interface, shown in line (3), is a list of distinct variables $\texttt{X}_1\texttt{,...,X}_n$ which are the arguments to the CHR constraint. Line (4) allocates a new constraint identifier. Line (5) constructs the constraint `C` ready for insertion into the store, whereas line (6) actually does the insertion. Line (7) sets up any necessary delayed goals on any solver variables in `C`. Notice that the same identifier `Id` is used for setting up the delayed goals and as the constraint identifier. Finally, the constraint becomes active in line (8) by calling the first occurrence predicate `p_1`.

4.4.2 Occurrence Predicates

The main purpose of an occurrence predicate is to implement the behaviour of the **Simplify** or **Propagate** transitions from the refined operational semantics. For all CHR constraints p/n from the CHR program, and all occurrences m of p/n, the CHR compiler generates an occurrence predicate by the name p_m. This is usually the bulk of the code generated by the compiler.

The skeleton of an occurrence predicate (for the i^{th} occurrence) is shown in Figure 4.2. Lines (1)-(2) show the `pred` and `mode` declarations, which are identical to the same declarations for the top-level predicate, except for an extra argument for the constraint identifier (which we assume has type '`id`'). Line (3) is the interface, where the first argument is the constraint's identifier, followed by the constraint's actual arguments. Note that all of $\texttt{X}_1\texttt{,...,X}_n$ are the same set of distinct variables (because of normalisation) from the original rule. Line (4)

```
:- pred p_i(id,t_1,...,t_n).                                    (1)
:- mode p_i(in,m_1,...,m_n) is d.                               (2)
p_i(Id,X_1,...,X_n) :-                                          (3)
    <find-matches-and-call-body>                                (4)
    ( alive(Id) ->                                              (5)
        p_(i+1)(Id,X_1,...,X_n)                                 (6)
    ;   true                                                    (7)
    ).
```

Figure 4.2: Pseudo code for the i^{th} occurrence predicate.

represents the main purpose of this predicate, which is to find matching partners and fire the rule. We leave this part for now. After line (4) we can assume that all possible matchings have been tried. The rest of the occurrence predicate, lines (5)-(7) decides whether or not the next occurrence predicate, `p_`$(i+1)$, should be called. Line (5) tests if the active constraint has been deleted. If not then the next occurrence predicate is called in line (6) (in effect we are applying **Default** to the active constraint), otherwise the active constraint is effectively **Drop**ped in line (7). Note that for the last occurrence lines (5)-(7) are omitted.

The rest of this section is concerned with compiling the join, and firing the rule. Thanks to head and guard normalisation the head of the rule contains CHR constraints with only variable arguments, and none of those variables are repeated anywhere else in the head. This simplifies the problem of finding partner constraints considerably, since all constraints in the store with the same functor/arity as the partner constraint will always potentially match. The job of selecting which of these matches are valid is now entirely decided by the guard. The disadvantage of this approach is efficiency, but we (as usual) will postpone this problem until the optimisation chapter.

Because of program normalisation the CHR compiler has a list of partner constraints for a given active constraint and occurrence. We can break down the problem of finding a match (for the rule) into the problem of finding a match for individual partner constraints. Basically, given an iterator, we iterate through all potential matches for the first partner, and if a potential match is found, then we iterate through all potential matches for the second partner, and so on. If all partners have been matched, then the rule may fire provided the guard and propagation history tests succeed. After the rule fires, or after it failed to fire (e.g. if the guard failed), we return to the iterator for the last partner to find a new match. Either we find a new match for the last partner, or we return to the iterator for the previous partner, and so on. This system of iteration avoids the need to start the search for a set of matching constraints from scratch each time the rule fires, hence we avoid redundant work.

The code generated by the compiler that attempts to find a match for an individual partner is contained within a special *join-loop* predicate. Basically

a join-loop predicate iterates through all potential matches for some partner. If/when a matching is found, we either call the next join-loop predicate if there are more partners, or we call the *call-body* predicate which does the final checks before firing the rule. Likewise, execution only returns to a join-loop predicate once all matches for the remainder of the join have been tried.

For the implementation of a join-loop predicate there are complications to consider. Firstly, the operational semantics of CHRs disallows the same constraint from the store to be matched against more than one partner.[4] Secondly, the partner constraints that we select from the iterator may have been deleted since the original call to `get_iterator`. The problem arises because `get_iterator` is called once, then we iterate through the list, possibly firing the rule as we go. Unfortunately, it is possible that whilst firing the rule we "delete" some of the constraints in our iterator. Note that deletion removes the constraint from the global CHR store, and not from any (local) iterators, which are in effect a copy of an older store. Similarly, it is possible that any constraint from our partial match has been deleted in a similar fashion.

If deletion is a problem with iterators then it seems that insertion may also be a problem, i.e. whilst firing a rule we create new constraints that are potential matches. In fact there is no problem because of the call-based behaviour of the refined operational semantics. Any new CHR constraints created by a rule must have finished being active before we return from the firing of the rule. This means that all matches which include the new constraint must have already been tried, so there is no need to update the iterators. Actually this is another advantage of using iterators, since we never consider these constraints we save redundant work.

The pseudo code for a join-loop predicate is shown in Figure 4.3. Here we assume that for an active `p` constraint (at the i^{th} occurrence) we are trying to find matching partner `q`(A, B, C) (the code is similar for any other constraint). Lines (1)-(2) show the `pred` and `mode` declarations. Line (3) handles the case of an empty iterator, where no action is necessary. Line (4) handles the other case, and also demonstrates the interface for a join-loop predicate. The first argument is the iterator itself, which we assume has type '`iterator`' defined as follows:

```
:- typedef iterator = list(numbered).
:- typedef numbered -> constraint # id.
```

This means an iterator is a list of numbered constraints, where type '`constraint`' is the type given to the constraints in the CHR store. Next is $\texttt{Id}_1\texttt{,...,Id}_n$ which are the constraint identifiers from the active constraint and from any matches of partners so far (the partial match). Similarly, $\texttt{X}_1\texttt{,...,X}_k$ are the combined arguments from the partial match. Here we assume that $t_1, ..., t_k$ and $m_1, ..., m_k$ are the types and modes associated with $\texttt{X}_1\texttt{,...,X}_k$ (based on the types and

[4]This was the purpose of the multiset union $\uplus$ in the specification of **Simplify** and **Propagate**.

```
:- pred p_i_join_loop(iterator,id,...,id,t_1,...,t_k).                (1)
:- mode p_i_join_loop(in      ,in,...,in,m_1,...,m_k) is d.           (2)
p_i_join_loop([],_,...,_).                                            (3)
p_i_join_loop([C # Id|Ls],Id_1,...,Id_n,X_1,...,X_k) :-               (4)
    (
      C = q(A,B,C),                                                   (5)
      alive(Id),                                                      (6)
      Id \= Id_1,                                                     (7)
      ⋮
      Id \= Id_n ->                                                   (8)
        <find-matches-and-call-body>                                  (9)
    ;   true                                                          (10)
    ),
    (
      alive(Id_1),                                                    (11)
      ⋮
      alive(Id_n) ->                                                  (12)
        p_i_join_loop(Ls,Id_1,...,Id_n,X_1,...,X_k)                   (13)
    ;   true                                                          (14)
    ).
```

Figure 4.3: Pseudo code for the join loop predicate.

modes of the constraints involved in the partial match). In line (4) we have also extracted a new constraint `C` with identifier `Id` from the iterator.

The rest of the code is split into two parts. The first part, lines (5)-(10), decides if the constraint `C` matches against the partner (i.e. if `C` is a `q`/3 constraint), and takes appropriate action if so. Line (5) deconstructs the newly acquired constraint `C`, and fails if it is not a `q`/3. Line (6) tests if the identifier `Id` has not been deleted (recall that deletion is possible). Lines (7)-(8) test if the new identifier `Id` has not been matched earlier in this join (this is to ensure multiset matching). If all of these tests pass, then in line (9) we call the code for finding a match for the next partner if necessary, or the call the call-body predicate otherwise.

The second part of the join-loop predicate, in lines (11)-(14), decides if we are allowed to continue looking for partners, or if we must abort because a constraint from our partial match has been deleted. The invariant we are maintaining is that all constraints in the partial match have not been deleted. Lines (11)-(12) test if all the identifiers from the partial matching are still alive. If so then we recursively call the join-loop predicate with the same arguments except with the tail of the original iterator, otherwise no action is necessary.

We can now give a simple example of a join-loop predicate.

Example 35 *Consider occurrence number six for the* `leq` *constraint in the fol-*

```
:- pred leq_6_join_loop(iterator,id,hint,hint).
:- mode leq_6_join_loop(in,in,in,in) is semidet.
leq_6_join_loop([],_,_,_).
leq_6_join_loop([C#J|Ls],I,X,Y) :-
    (
      C = leq(W,Z),
      alive(J),
      J \= I ->
        leq_6_call_body(I,J,X,Y,W,Z)
    ;   true
    ),
    (
      alive(I) ->
        leq_6_join_loop(Ls,I,X,Y)
    ;   true
    ).
```

Figure 4.4: Join loop predicate for the transitivity rule's first occurrence.

lowing (normalised) rule.

$\texttt{leq(X,Y)}_6$, `leq(W,Z) ==> Y = W | leq(X,Z).`

To find matching constraints for the partner ***`leq(W,Z)`*** *the compiler generates the join-loop predicate as shown in Figure 4.4. Here, we assume that the type of the arguments to the* ***`leq`*** *constraints is* ***`hint`*** *(Herbrand* ***`int`****).* □

After a set of matching constraints has been found we need to check the guard and propagation history, then delete any constraints if necessary, before calling the code in the body. This is the role of the *body-call* predicate, whose pseudo code is shown in Figure 4.5. Lines (1)-(2) are the `pred` and `mode` declarations. The interface to the body call predicate, shown by line (3), is very similar to that of the join-loop predicate. All of $\texttt{Id}_1,\ldots,\texttt{Id}_n$ are constraint identifiers, and all of $\texttt{X}_1,\ldots,\texttt{X}_k$ are arguments of the matching respectively. Note that the interface could be improved by eliminating arguments that are not actually required by the predicate's body.

Line (4) tests the guard and line (5) checks the propagation history. Exactly how the guard is tested is left for the next section. Checking the propagation history involves constructing an entry and then calling `check_history` described previously. The history check is omitted whenever rule r is not a propagation rule. The order of the identifiers in the history entry is defined by the auxiliary function *order*, which is evaluated at compile time. We assume that function *order* sorts the identifiers based on the textual order of the constraints in the rule head. It is important that all call-body predicates use the same order for the same rule.

```
:- pred p_i_call_body(id,...,id,t_1,...,t_k).                    (1)
:- mode p_i_call_body(in,...,in,m_1,...,m_k) is d.               (2)
p_i_call_body(Id_1,...,Id_n,X_1,...,X_k) :-                      (3)
    (
      <test-guard>,                                              (4)
      check_history([order(Id_1,...,Id_n),r]) ->                 (5)
        delete(Id'_1),                                           (6)
        ⋮
        delete(Id'_j),                                           (7)
        <body>                                                   (8)
    ;   true
    ).
```

Figure 4.5: Pseudo code for the call body predicate.

If both the guard test and history check pass then the rule can fire. If the rule is not a propagation rule then some of the matching constraints need to be deleted. Let $\{\texttt{Id}'_1,\ldots,\texttt{Id}'_j\} \subseteq \{\texttt{Id}_1,\ldots,\texttt{Id}_n\}$ be the identifiers of these constraints, then lines (6)-(7) explicitly do the deletions. Finally, the body of the rule is called in line (6). The body is usually copied verbatim from the original rule.

We are now ready for a complete example.

Example 36 (Compiled `gcd` Program) *The compiled version of the* `gcd` *program is given in Figure 4.6. Note that we gloss over guard compilation (the guards are inserted "as-is" into the compiled code) for the time being. Also, we omit the* `pred` *and* `mode` *declarations for brevity. The new program consists of one top-level predicate, three occurrence predicates, two join-loop predicates and three body-call predicates.* □

The compiled version of the `gcd` program is much larger than the original CHR version, and this is generally true for all CHR compilation. Several improvements are possible, e.g. inlining the body-call predicates, and inlining the first occurrence predicate with the top-level predicate. After optimisation, the resulting code will be considerably more compact.

4.5 Compiling the Guard

Let r be a normalised CHR rule with guard g. The operational semantics of CHRs dictates that (a renamed copy of) r can only fire iff $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g)$ holds, where B is the current builti-n store, θ is the matching substitution, and the (possibly incomplete) test $(\mathcal{D} \models_{\mathcal{S}})$ (see Definition 4) represents the solver.

In practice the built-in solver must not only provide a procedure for *telling* a constraint (adding it to the built-in store) whenever it appears in the body of

```
gcd(X) :-
    C = gcd(X),
    new(I),
    insert(C,I),
    delay(C,I,gcd_1(I,X)),
    gcd_1(I,X).

gcd_1(I,X) :-
    gcd_1_call_body(I,X),
    ( alive(I) ->
        gcd_2(I,X)
    ;   true
    ).

gcd_1_call_body(I,X) :-
    ( X = 0 ->
        delete(I)
    ;   true
    ).

gcd_2(I,N) :-
    get_iterator(Ls),
    gcd_2_join_loop(Ls,I,N),
    ( alive(I) ->
        gcd_3(I,X)
    ;   true
    ).

gcd_2_join_loop([],_,_).
gcd_2_join_loop([C # J|Ls],I,N) :-
    (
      C = gcd(M),
      alive(J),
      J \= I ->
        gcd_2_call_body(I,N,M)
    ;   true
    ),
    ( alive(I) ->
        gcd_2_join_loop(Ls,I,N)
    ;   true
    ).

gcd_2_call_body(I,N,M) :-
    ( M =< N ->
        delete(I),
        gcd(M-N)
    ;   true
    ).

gcd_3(I,M) :-
    get_iterator(Ls),
    gcd_3_join_loop(Ls,I,M).

gcd_3_join_loop([],_,_).
gcd_3_join_loop([C # J|Ls],I,M) :-
    (
      C = gcd(N),
      alive(J),
      J \= I ->
        gcd_3_call_body(J,N,M)
    ;   true
    ),
    ( alive(I) ->
        gcd_3_join_loop(Ls,I,M)
    ;   true
    ).

gcd_3_call_body(J,N,M) :-
    ( M =< N ->
        delete(J),
        gcd(M-N)
    ;   true
    ).
```

Figure 4.6: Compiled version of the `gcd` program.

the rule, but also a procedure for *asking* a constraint (determining if the guard is entailed by the current built-in store) whenever it appears in the guard of the rule. Formally an ask constraint c holds iff $\mathcal{D} \models_{\mathcal{S}} B \rightarrow c$. For example, in the case of the Herbrand solver, the only tell constraint (`=/2`) has the known associated ask constraint `==/2`.

This section presents how guard entailment is implemented in the HAL CHR compiler. We will first assume that the guard g contains no existentially quantified variables (i.e., all variables in the guard also appear in the head). Existentially quantified variables in the guard are dealt with in the next subsection.

4.5.1 Basic Guards

Solvers in HAL must define a type for the solver variables (e.g., `fdint`), and code to initialise new solver variables. Often a solver variable will be some form of pointer into a global store of variable information. The solver also defines predicates for the constraints supported by the solver. These predicates define *tell* constraints for that solver (e.g., they provide the code for predicate `X =< Y` which adds constraint $X \leq Y$ to the solvers store). Often the solver is also packaged as an appropriate instance of a solver type class, and thus the solver is known to provide at least the constraints included in the type class interface. For more details on HAL solver classes see e.g., [17].

In order for a constraint solver to be extended by CHRs, the solver needs to provide *ask* versions of the constraints that it supports. It is the ask version of the constraints that should be used in guards.

Example 37 *For example, consider the following CHR program that implements a* `min`/3 *constraint over a finite domain solver.*

```
min(A,B,C) <=> A =< B | C = A.
min(A,B,C) <=> A >= B | C = B.
```

Consider the compilation of the first rule. The constraint `A =< B` *will be replaced by some predicate which implements the ask version of the finite domain* `=<` *constraint, e.g,* `'ask_=<'(A,B)`. □

Every CHR implementation that we are aware of automatically translates tell to ask constraints in the guards. However, these implementations only deal with one built-in solver (Herbrand). When arbitrary solvers are used, the compiler needs a general method for determining the relationship between the tell and ask versions of each constraint so that it can automatically transform one into the other. In HAL this general method is achieved by the following `asks` declaration.

`:-` <*ask-constraint*> `asks` <*tell-constraint*>`.`

which defines a mapping from a tell to an ask constraint.

The `asks` declaration is effectively a macro definition on which the following restrictions apply. First, each tell constraint can only have one associated ask

constraint (although an ask constraint can be associated to more than one tell). Second, the arguments of the *tell-constraint* must be distinct variables, and only these and anonymous variables can appear in the corresponding *ask-constraint*. And finally, the ask constraint must be defined for the type of arguments of the associated tell constraints, it must be usable in any mode in which the associated tell constraints are, and must have `semidet` determinism.

Example 38 *The following are* `asks` *declarations that might be declared by a finite domain solver.*

```
:- 'ask_=<'(X,Y) asks X =< Y.
:- 'ask_=<'(Y,X) asks X >= Y.
```

The CHR compiler uses this information to translate the guards from Example 37 into appropriate ask constraints. □

A predicate is recognised by the compiler as a tell constraint iff it has been declared as having an associated ask constraint. The compiler automatically replaces each such tell constraint which textually appears in a guard with its ask version. In addition, HAL (and Prolog CHR implementations) also allow arbitrary predicates in the guard. This means that tell constraints nested inside the guard will be treated as tells, when perhaps this was not the intention of the programmer. The HAL compiler warns if this can occur.

4.5.2 Guards with Existential Variables

So far guard entailment testing only deals with the case where the variables in the guard appear in the rule head. All other variables in the guard being existentially quantified. Existential quantification can change the operational behaviour of an ask constraint, hence guards with existential variables must be specially dealt with.

Example 39 *Consider the ask constraint $X = Y$ which asks if two variables X and Y are equal. Obviously, this ask constraint may fail if X and Y differ. Now consider a "similar" looking ask constraint $\exists X.X = Y$, where X is now existentially quantified. This ask constraints always (trivially) succeeds, since setting X to be equal to Y is a solution.* □

Many existential variables arise from normalisation due to limitations of the built-in solver.

Example 40 *Consider the following CHR constraint* `before_after` *used in task scheduling which extends a finite domain solver. Basically, the constraint* `before_after`*($T1$,$D1$,$T2$,$D2$) holds if the task with start time $T1$ and duration $D1$ does not overlap with the task with start time $T2$ and duration $D2$.*

```
before_after(T1,D1,T2,D2) <=> T1 + D1 > T2 | T1 >= T2 + D2.
before_after(T1,D1,T2,D2) <=> T2 + D2 > T1 | T2 >= T1 + D1.
```

Normalisation of the guard results in:

```
before_after(T1,D1,T2,D2) <=> exists [N] (N = T1 + D1, N > T2) |
                                         T1 >= T2 + D2.
before_after(T1,D1,T2,D2) <=> exists [N] (N = T2 + D2, N > T1) |
                                         T2 >= T1 + D1.
```

thus introducing a new existential variable N. □

Unfortunately, it is difficult to automatically handle ask constraints with existential variables. Only the built-in solver can answer general questions about existential guards.

Example 41 *Consider an integer solver which supports the constraint* `X =< Y`*. Consider the following (somewhat contrived) rule for a CHR constraint* ***my_fail***

```
my_fail(X) <=> X =< N, N =< X | fail.
```

The logical reading of the guard is $\exists N(X \leq N \wedge N \leq X)$ which always holds (e.g., try $N = X$). Hence, a ***my_fail*** *constraint should reduce to the built-in* ***fail****. However, it is impossible to separate the two primitive constraints occurring in the ask formula into two independent ask constraints. Instead, we would have to ask the solver to treat the entire conjunction at once.* □

There is always a programming solution to handling guards with general existential variables. For example, the CHR compiler could reject the rule from Example 41 (e.g. by throwing a compiler error), thereby forcing the programmer to rewrite the guard and remove the existential variable. More generally, the programmer may have to redesign the solver itself in order to handle more complicated ask constraints, including conjunctions of ask constraints. Obviously, this solution is not very practical, and we expect the CHR compiler to be able to handle at least some subset of all guards with existential variables.

Although normalisation can lead to proliferation of existential variables in the guard, in many cases such existential variables can be compiled away without requiring extra help from the solver. Consider determining whether a constraint store B implies a guard g of the following form

$$v = f(y_1, \ldots, y_n) \wedge g'$$

Where g' represents the rest of the guard, and variable v is existentially quantified. Let $\bar{x}$ be the list of existentially quantified variables excluding v. If f is a total function, such a v always exists and is unique, then as long as none of the variables $y_1, \ldots, y_n$ are existentially quantified (i.e. appear in $\bar{x}$) then the question $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists \bar{x} \exists v.g$ is equivalent[5] to the question

$$\mathcal{D} \models_{\mathcal{S}} (B \wedge v = f(y_1, \ldots, y_n)) \rightarrow (\exists \bar{x} g')$$

[5]For incomplete solvers, equivalence is effectively a condition on $(\mathcal{D} \models_{\mathcal{S}})$. In HAL, where logical connectives (e.g. $\wedge$) are handled by the compiler rather than the built-in solver, we can assume that equivalence holds.

Now v is no longer existential in g'.

This translation gives the first hint of how to compile guards with existential quantification. Basically the formula $v = f(y_1, \ldots, y_n)$ can be compiled to a tell constraint which constrains a (fresh) variable v to be the result of applying function f to $y_1, \ldots, y_n$, followed by the code for the rest of the guard g'. Note that order is now important, we execute the tell constraint $v = f(y_1, \ldots, y_n)$ before g'.

Example 42 *Returning to Example 40. The call-body predicate (including the guard test) for the first rule is as follows.*

```
before_after_1_call_body(I,T1,D1,T2,D2) :-
    (
      N = T1 + D1,                      % TELL constraint
      'ask_>'(N,T2) ->                  % ASK constraint
        delete(I),
        T1 >= T2 + D2                   % Body
    ;   true
    ).
```

Here we assume that the ask version of the (<$/2$) constraint is `'ask_<'`$/2$ *with the same arguments. In the first rule the ask constraint* $\exists N(N = T1 + D1)$ *becomes a tell constraint, because neither* `T1` *nor* `D1` *are existentially quantified, and the function* `+` *is total. The same reasoning applies to the second rule.* □

There are other common cases that allow us to compile away existential variables, but require some support from the built-in solver. Consider a guard g of the same form as before:

$$v = f(y_1, \ldots, y_n) \wedge g'$$

Where g' represents the rest of the guard, variable v is existentially quantified and none of $y_1, \ldots, y_n$ are existentially quantified, except this time f is a *partial* function. Then the question $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists \bar{x} \exists v.g$ is equivalent to

$$\mathcal{D} \models_{\mathcal{S}} (B \rightarrow \exists w.w = f(y_1, ..., y_n)) \wedge ((B \wedge v = f(y_1, ..., y_n)) \rightarrow (\exists \bar{x} g'))$$

since if there exists a w of the form $f(y_1, \ldots, y_n)$, then it is unique. Hence, the function f in the context of this test is effectively total, and can thus become a tell constraint. This may not seem to be a simplification, but if we provide an ask version of the constraint $\exists w, w = f(y_1, \ldots, y_n)$ then we can indeed handle partial functions in the guard.

Example 43 *Consider the rule*

```
g(X,Y,B) <=> X + 2^Y ≥ 2 | gg(X,Y,B)
```

which is normalised to

```
g(X,Y,B) <=> exists [Z,N,M] (Z = 2^Y, N = X + Z, M = 2, N ≥ M) |
                gg(X,Y,B).
```

If the compiler knows that the ask version of the constraint $\exists Z.Z = 2^Y$ is equivalent to ***nonneg(Y)*** *(which holds if Y is not negative), then the rule has the following call-body predicate*

```
g_1_call_body(I,X,Y,B) :-
    (
      nonneg(Y),                        % ASK constraint
      Z = 2^Y,                          % TELL constraint
      N = X + Z,                        % TELL constraint
      M = 2,                            % TELL constraint
      'ask_>='(N,M) ->                  % ASK constraint
        delete(I),
        gg(X,Y,B)
    ;   true
    ).
```

The constraint $Z = 2^Y$ is, in effect, a total function (since we know that $Y \geq 0$), and is thus compiled as a tell constraint. □

To use this simplification we need ask versions of constraints for partial functions. These can be provided using the already introduced mechanisms for mapping tell constraints to ask constraints.

Example 44 *For example, the mapping for $z = 2^y$ for a finite domain solver can be defined as*

```
:- nonneg(Y) asks exists [Z] Z = 2^Y.
```

□

To apply either of the simplifications above we also require information about total and partial functions. The HAL compiler already receives this information from the solver by means of mode declarations. For example, the following mode declarations

```
:- mode oo + oo --> no is det.
:- mode oo ^ oo --> no is semidet.
```

show the totality of function `+` (since the function is declared to be `det`, i.e. have one and only one solution for every two input arguments) and the partialness of function `^` (since the function is declared to be `semidet`, i.e. have either one solution or none for every two input arguments). Since functions in HAL are really predicates with an additional argument, it is straightforward to extend these simplifications to use predicate rather than functional notation.

Example 45 *Consider the constraint predicate* ***munion(A,B,C)*** *which constrains C to be the multiset resulting from the union of the two multisets A and B. Consider also that the following declarations for* ***munion*** *are provided:*

```
:- mode munion(oo,no,oo) is semidet.
:- msub(A,C) asks exists [B] munion(A,B,C).
```

indicating a partial function behaviour, and its corresponding asks-test. Also consider the following normalised rule.

```
h(A,C,D) <=> exists [B] (munion(A,B,C), B ≠ D) | hh(A,D).
```

The call-body predicate for this rule is

```
h_1_call_body(I,A,C,D) :-
    (
      msub(A,C),                        % ASK constraint
      munion(A,B,C),                    % TELL constraint
      ask_neq(B,D) ->                   % ASK constraint
        delete(I),
        hh(A,D)
    ;   true
    ).
```

□

Partial functions are common in Herbrand constraints. Consider the constraint $x = f(y_1, \ldots, y_n)$, where f is a Herbrand constructor. This constraint defines, among others, a partial (deconstruct) function f_i^{-1} from x to each $y_i, 1 \leq i \leq n$. For this reason the compiler produces new ask tests `bound_f(X)` for each Herbrand constructor f, which check whether X is bound to the function f. Herbrand term deconstructions are then compiled as if the asks declaration

```
:- bound_f(X) asks exists [Y1,..,Yn] X = f(Y1,..,Yn).
```

appeared in the program.

Example 46 *Consider the compilation of the guard from Example 32.*

```
length(Xs1,L) <=> exists [A,Xs] (Xs1 = [A|Xs]) |
                  L = L1+1, length(Xs,L1).
```

We will compile this guard to the call ***'bound_[|]'(Xs1)****.* □

It is interesting to note that tell constraints in guards mean that each time the guard is tested, new tell constraints must be added to the built-in store. In theory this is no problem, since the new constraints do not effect the logical meaning of the store, and can safely be projected out once the guard has been

tested (hence we refer to these constraints as *temporary* constraints). In practice however, projection is typically not done, and temporary constraints remain in the built-in store for the rest of the derivation. This may be inefficient, since each constraint typically consumes resources (e.g. memory).

Example 47 *Consider the following CHR extending a finite domain solver.*

```
p(X,Y), q(Y,Z) ==> X + Z < Y | s(Z).
```

which normalised and simplified gives

```
p(X,Y), q(Y1,Z) ==> exists [W] (Y = Y1, W = X + Z, W < Y) | s(Z).
```

For an active `p(X,Y)` *constraint, the finite domain constraint* $W = X + Z$ *(which will be compiled as a tell constraint) will be added to the store for each matching* `q(Y,Z)` *currently in the CHR store. If the (overall) guard succeeds* k *times then* k *copies will remain in the store.* □

This problem is not new, nor CHR-specific. Any CLP program that creates temporary variables/constraints may suffer from a similar inefficiency. Some solutions already exist, for example [58] proposes a program analysis that explicitly projects out "dead" (temporary) variables once they are no longer needed. If a HAL solver supported projection, and we performed the associated dead variable elimination analysis, then this could be applied to the output of the CHR compiler, solving the problem.

Not all solvers suffer terribly from this problem. Garbage collection for Herbrand solvers will remove dead variables and their associated constraints. Currently the HAL CHR compiler does nothing to remove temporary constraints and variables arising from asks. We leave improvement to future work. Fortunately, many CHR programs (including the key examples from this thesis) are not affected by this problem.

4.6 Summary

In this chapter we have described how to compile a CHR program into a logic programming language, specifically HAL. Like most other programming languages, compiling CHRs in a multi-phase process, including phases for parsing and normalisation, guard compilation and code generation, which were all covered here. Also, a reasonably simple runtime environment was described, which implements the execution state for the refined semantics.

Parsing CHR rules is not much different than parsing any other programming language. The only difference is that as CHRs are usually an embedded language, the job of parsing CHRs is left to the parser for the host language. Normalisation is an important as it helps simplify later compilation phases. With CHRs we have identified two distinct types of normalisation: guard normalisation and program normalisation. Guard normalisation helps significantly simplify the compilation

of the guard later, and program normalisation is especially useful for code generation.

Guard compilation is a surprisingly involved aspect of CHR compilation. This is for several reasons. Firstly, in order to make compilation work for arbitrary built-in solvers a general solver interface was devised, namely mapping tell constraints to ask constraints via a special `asks` declaration. Guards with existentially quantified variables occur often in practice, so a CHR compiler should be able to handle them. Existential variables which are functionally defined by non-existential variables in the guard can be compiled without any additional help from the solver. Unfortunately, a more general solution would require a more complicated interface with the built-in solver.

With the program normalised and the guard compiled, the final phase of CHR compilation is code generation. Many aspects of the CHR execution state map neatly into similar concepts in the HAL language, e.g. the execution stack becomes the program stack, and the built-in store is implicit as ordinary HAL solvers.

The output of the CHR compiler is a series of HAL predicates that manipulate a global CHR constraint store. The CHR compiler replaces each constraint with a so-called *top-level* predicate that performs the necessary overhead before calling the first occurrence. Later, the *occurrence* predicates actually do the rule matching. This mostly involves iterating through the store for matches to individual partners to the active constraint, and this is the role of the *join-loop* predicates. Finally, once potential matches are found, the *call-body* predicate tests the guard and checks the history before actually firing the rule.

The compilation scheme presented in this chapter is very simple, and should give reasonable performance provided the rules themselves have small heads (no more than two heads) and the CHR store never grows too large at runtime. If either of these conditions do not hold then the result is likely to be a very inefficient program. Fortunately many improvements can be made which should improve performance somewhat and this is the focus for much of the rest of this thesis.

Chapter 5

Analysis

5.1 Introduction

Program analysis is a very important phase in CHR compilation. It is where we discover useful information about the program, which is mainly used for optimisation and (in the case of CHRs) confluence testing.

All of the program analysis presented in this chapter is based on abstract interpretation [16]. Abstract interpretation is a general methodology for program analysis by abstractly executing the program code. It provides the remedy for the current difficulties in correctly analysing CHR programs, and should enable optimising CHR compilers to reach a new level of complexity and correctness.

We will present an abstract interpretation framework based on a slightly different formulation of the refined operational semantics. The new semantics is the *call-based operational semantics* for CHRs. The call-based semantics is equivalent to the refined semantics (and we provide results to that effect), however they help simplify the abstract interpretation framework.

We will present two main kinds of CHR analysis based on our abstract interpretation framework. These include

- Late storage analysis – decides when the active constraint should be stored;
- Functional dependency analysis – decides if there exist functional dependencies between arguments of CHR constraints.

Both of these analyses are relevant and useful to the rest of this thesis.

The rest of this chapter is structured as follows. First, Section 5.2 presents the call-based operational semantics of CHR that will be abstractly interpreted. Section 5.3 establishes equivalence of the call-based and refined operational semantics. Section 5.4 defines the general abstract interpretation framework. Sections 5.5 and 5.6 present two instances of the framework, implementing late storage and functional dependency analysis respectively. Finally, we conclude.

5.2 The Call-based Operational Semantics ω_c

In this section we define the *call-based* operational semantics for CHRs, which we shall refer to as ω_c. They are a variant of the refined operational semantics of Chapter 3, but are designed to make the analysis more straightforward. For the analysis of logic programs, we do not directly analyse over the derivations based operational semantics, instead we introduce a call based semantics which makes the number of abstract goals to be considered finite (see e.g. [60]). We introduce the call-based operational semantics for CHRs for exactly the same reason.

The main difference between the two semantics lies in their formulation. The transition system of ω_r linearises the dynamic call-graph of CHR constraints into the execution stack. On the other hand, under ω_c constraints are treated as procedure calls: each new *active* constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires other CHR constraints may be executed as subcomputations and, when they finish, the execution returns to finding rules for the current active constraint. The latter semantics is much closer to the procedure-based target languages, like Prolog and HAL, of the current CHR compilers.[1]

Much of the basic framework for the refined semantics still applies under the call-based semantics. For example, the definition of built-in constraints, CHR store, propagation history, etc. is exactly the same as before. Likewise the definition of $fixed(B)$ (the set of variables fixed by B), and the *wakeup policy* have not changed.

There are differences between the two semantics. For example, the definition of an *execution state* has changed.

Definition 22 (Call-based execution state) *Formally, the execution state of the call-based semantics is the tuple $\langle G, A, S, B, T\rangle_n^{\mathcal{V}}$ where G is the current goal, A is a sequence of active constraints (called the* activation stack*), S is a set of numbered constraints, B is a conjunction of built-in constraints, T is a set of sequences of integers, $\mathcal{V}$ is the set of variables and n is an integer. The goal is either a single constraint, or a sequence of built-in or CHR constraints.* □

The role of S, B, T, $\mathcal{V}$ and n is the same as in the refined semantics. The goal G roughly corresponds to the top of a refined execution stack, and may be any type of constraint, e.g. built-in, CHR, active, etc. The activation stack A roughly corresponds to the tail of the execution stack under the refined semantics, except it only contains active constraints.

Given initial goal G, the initial state is

$$\langle G, [], \emptyset, true, \emptyset\rangle_1^{vars(G)}$$

[1] It can be argued that the basic compilation is more closely related to the call-based semantics, since each active constraint is compiled into a procedure call.

Execution proceeds by exhaustively applying transitions to the initial execution state until the built-in solver state is unsatisfiable or no transitions are applicable.

The possible transitions for the call-based semantics are as follows:

Definition 23 (Call-based Operational Semantics)
1. Solve

$$\langle c, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \Box, A, S', B', T'\rangle_{n'}^{\mathcal{V}}$$

where c is a built-in constraint. If $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset} c \wedge B$, then $S' = S$, $B' = c \wedge B$, $T' = T$, $n' = n$. Otherwise ($\mathcal{D} \models_{\mathcal{S}} \bar{\exists}_{\emptyset} c \wedge B$), where

$$\langle wakeup\_policy(S, c, B), A, S, c \wedge B, T\rangle_n^{\mathcal{V}} \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}^{\mathcal{V}} \tag{5.1}$$

Note that the *wakeup_policy* is a function satisfying Definition 10, just as in the refined semantics.
2. Activate

$$\langle c, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle c\#n : 1, A, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}^{\mathcal{V}}$$

where c is a CHR constraint (which has never been active).
3. Reactivate

$$\langle c\#i, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle c\#i : 1, A, S, B, T\rangle_n^{\mathcal{V}}$$

where c is a CHR constraint.
4. Drop

$$\langle c\#i : j, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \Box, A, S, B, T\rangle_n^{\mathcal{V}}$$

where $c\#i : j$ is an active constraint and there is no such occurrence j in P.
5. Simplify

$$\langle c\#i : j, A, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \Box, A, S', B', T''\rangle_{n'}^{\mathcal{V}}$$

where

$$\langle \theta(C), A, H_1 \uplus S, B, T'\rangle_n^{\mathcal{V}} \rightarrowtail^* \langle \Box, A, S', B', T''\rangle_{n'}^{\mathcal{V}} \tag{5.2}$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

and there exists matching substitution θ is such that

$$\begin{cases} c = \theta(d_j) \\ cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ cons(H_3) = \theta(H_3') \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

In the intermediate transition sequence $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.

If no such matching substitution θ exists then

$$\langle c\#i : j, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle c\#i : j+1, A, S, B, T\rangle_n^{\mathcal{V}}$$

6. Propagate

$$\langle c\#i : j, A, \{c\#i\} \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle G, A, S_k, B_k, T_k\rangle_{n_k}^{\mathcal{V}}$$

where the j^{th} occurrence of the CHR predicate of c in a rule in P is

$$r \ @ \ H_1', d_j, H_2' \setminus H_3' \Longleftrightarrow g \mid C$$

Let $S_0 = S \uplus \{c\#i\}, B_0 = B, T_0 = T, n_0 = n$. Now assume, for $1 \leq l \leq k$ and $k \geq 0$, the series of transitions

$$\begin{array}{c}\langle \theta_l(C_l), [c\#i : j|A], H_{1l} \uplus \{c\#i\} \uplus H_{2l} \uplus S'_{l-1}, B_{l-1}, T_{l-1} \cup \{t_l\}\rangle_{n_{l-1}}^{\mathcal{V}} \\ \rightarrowtail^* \langle \Box, [c\#i : j|A], S_l, B_l, T_l\rangle_{n_l}^{\mathcal{V}}\end{array} \tag{5.3}$$

where $\{c\#i\} \uplus H_{1l} \uplus H_{2l} \uplus H_{3l} \uplus S'_{l-1} = S_{l-1}$ and there exists a matching substitution θ_l, and a renamed copy of rule r

$$r \ @ \ H_{1l}', d_{jl}, H_{2l}' \setminus H_{3l}' \Longleftrightarrow g_l \mid C_l$$

such that

$$\begin{cases} c = \theta_l(d_{jl}) \\ cons(H_{1l}) = \theta_l(H_{1l}') \\ cons(H_{2l}) = \theta_l(H_{2l}') \\ cons(H_{3l}) = \theta_l(H_{3l}') \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta_l \wedge g_l) \\ t_l = id(H_{1l}) +\!\!+ [i] +\!\!+ id(H_{2l}) +\!\!+ id(H_{3l}) +\!\!+ [r] \notin T_{l-1} \end{cases}$$

Furthermore, for $k+1$ no such transition is possible.

The resulting goal G is either $G = \Box$ if $\mathcal{D} \models_{\mathcal{S}} \bar{\exists}_{\emptyset}(\neg B_k)$ (i.e. failure occurred) or $G = c\#i : j+1$ otherwise.

7. Goal

$$\langle [c|C], A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle G, A, S', B', T'\rangle_{n'}^{\mathcal{V}}$$

where $[c|C]$ is a sequence of built-in and CHR constraints

$$\langle c, A, S, B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'} \tag{5.4}$$

and $G = \Box$ if $\mathcal{D} \models_{\mathcal{S}} \bar{\exists}_{\emptyset} \neg B'$ (i.e. calling c caused failure) or $G = C$ otherwise. $\Box$

Some of the call-based transitions are very similar to the refined semantics version. For example, the **Activate**, **Reactivate** and **Drop** transitions are very similar. Other transitions, such as **Propagate**, are radically different. Also, the **Default** transition has disappeared, and a new transition **Goal** has appeared. The **Goal** transition is for executing sequences of constraints in order.

The main difference from the refined transitions is the notion of a *subcomputation*. A *subcomputation* is defined to be the subderivation used by the **Solve**, **Simplify**, **Propagate** or **Goal** transitions in order to calculate the final state. The subcomputations are marked (5.1), (5.2), (5.3) and (5.4) above in Definition 23. Subcomputations are the key to the call-based semantics, since they represent the *calling* (and execution) of a subgoal.

There is no **Default** transition under the call-based semantics. Instead the **Default** transition has effectively been merged into the **Simplify** and **Propagate** transitions. This means that once a rule cannot fire, these transitions automatically moves the active constraint to the next occurrence. This will simplify the abstract interpretation framework somewhat.

A sequence of **Propagate** transitions under the refined semantics has similarly been merged into a single **Propagate** transition under the call-based semantics. The new **Propagate** effectively tries all possible rule firings before moving to the next occurrence (see the sequence of subcomputation in (5.3)). This means that, unlike the refined semantics, after a call-based **Propagate** transition has finished, the propagation rule is no longer applicable to any matching including the active constraint for the given occurrence.

Now we illustrate the call-based semantics on a small example program:

Example 48 *Consider the following (somewhat contrived) CHR program:*

```
p1      ==> q.
p2, t1 <=> r.
p3, r1 ==> true.
p4      ==> s.
p5, s1 <=> true.
```

All the occurrences of constraints in the above program are indexed with their respective occurrence numbers. Starting from an initial goal `p` *the derivation under the call-based operational semantics is shown in Figure 5.1. Note that for brevity we omit the built-in store B, $\mathcal{V}$, and propagation history T.*

We have abbreviated ***Activate*** *with A,* ***Drop*** *with Dp,* ***Simplify*** *with Si and* ***Propagate*** *with P. For both the simplification Si transitions, we annotate the name with $\neg$ if the transition did not find a match (i.e. a* ***Default*** *transition under the refined semantics).* □

Whilst the call-based semantics is useful for abstract interpretation, for the rest of the thesis (excluding this chapter) we revert back to the refined semantics.

$$
\begin{array}{ll}
 & \langle p, [], \emptyset \rangle_1 \\
\rightarrowtail_A & \langle p\#1 : 1, [], \{p\#1\} \rangle_2 \\
\rightarrowtail_P & \langle p\#1 : 2, [], \{p\#1, q\#2\} \rangle_2 \\
 & \qquad \langle q, [p\#1 : 1], \{p\#1\} \rangle_2 \qquad (q'\text{s subcomputation}) \\
 & \rightarrowtail^* \langle \Box, [p\#1 : 1], \{p\#1, q\#2\} \rangle_3 \\
\rightarrowtail_{\neg Si} & \langle p\#1 : 3, [], \{p\#1, q\#2\} \rangle_3 \\
\rightarrowtail_{\neg P} & \langle p\#1 : 4, [], \{p\#1, q\#2\} \rangle_3 \\
\rightarrowtail_P & \langle p\#1 : 5, [], \{q\#2\} \rangle_4 \\
 & \qquad \langle s, [p\#1 : 4], \{p\#1, q\#2\} \rangle_3 \quad (s'\text{s subcomputation}) \\
 & \rightarrowtail^* \langle \Box, [p\#1 : 4], \{q\#2\} \rangle_4 \\
\rightarrowtail_{\neg Si} & \langle p\#1 : 6, [], \{q\#2\} \rangle_4 \\
\rightarrowtail_{Dp} & \langle \Box, [], \{q\#2\} \rangle_4
\end{array}
$$

q's subcomputation :

$$
\begin{array}{ll}
 & \langle q, [p\#1 : 1], \{p\#1\} \rangle_2 \\
\rightarrowtail_A & \langle q\#2 : 1, [p\#1 : 1], \{p\#1, q\#2\} \rangle_3 \\
\rightarrowtail_{Dp} & \langle \Box, [p\#1 : 1], \{p\#1, q\#2\} \rangle_3
\end{array}
$$

s's subcomputation :

$$
\begin{array}{ll}
 & \langle s, [p\#1 : 4], \{p\#1, q\#2\} \rangle_3 \\
\rightarrowtail_A & \langle s\#3 : 1, [p\#1 : 4], \{p\#1, q\#2, s\#3\} \rangle_4 \\
\rightarrowtail_{Si} & \langle \Box, [p\#1 : 4], \{q\#2\} \rangle_4 \\
 & \qquad \langle \Box, [s\#3 : 1, p\#1 : 4], \{p\#1, q\#2, s\#3\} \rangle_4 \\
 & \rightarrowtail^* \langle \Box, [s\#3 : 1, p\#1 : 4], \{p\#1, q\#2, s\#3\} \rangle_4
\end{array}
$$

Figure 5.1: Example derivation under the call-based operational semantics of CHRs

5.3 Equivalence of ω_c and ω_r

In this section we present a proof of equivalence between the call-based ω_c and refined ω_r operational semantics. This involves showing that a derivation under the call-based semantics can be mapped to an equivalent derivation under the refined semantics, and vice versa.

Note that a ω_c derivation does not always neatly map to an equivalent ω_r derivation because of the notion of a subcomputation. The proof is complicated by this fact.

5.3.1 From Call-based to Refined

In this subsection we show that some call-based derivations can be associated with equivalent refined derivations.

For simplicity, we will overload the sequence concatenation operator (++) as follows.

Definition 24 *The usual definition of ++ applies if both arguments are sequences, otherwise*

$$\begin{array}{ll} c \mathbin{+\!\!+} T & = [c|T] \qquad (c \textit{ constraint}) \\ \Box \mathbin{+\!\!+} T & = T \end{array}$$

□

The purpose is so we can easily construct a (refined) execution stack based on a call-based goal, which is not always a sequence. This modified definition of (++) will be implicitly used throughout the rest of this section.

Next we define the property that a ω_c derivation maps to an equivalent ω_r derivation. In which case we say that the ω_c derivation is *refined*.

Definition 25 (Refined) *We say a ω_c derivation*

$$\langle G, A, S, B, T\rangle_n \rightarrowtail^* \langle H, A, S', B', T'\rangle_{n'}$$

is refined *if for all A' the following are ω_r derivations (assuming the same wakeup policy is used):*

$$\langle G \mathbin{+\!\!+} A', S, B, T\rangle_n \rightarrowtail^* \langle H \mathbin{+\!\!+} A', S', B', T'\rangle_{n'}$$

□

The following lemma states that given a ω_c derivation (of a particular form) where each subcomputation is refined, is also refined.

Lemma 5 *All ω_c derivations*

$$\langle G, A, S, B, T\rangle_n \rightarrowtail^* \langle H, A, S', B', T'\rangle_{n'}$$

where all subcomputations are refined, are refined.

Proof. By induction.

Base Case: Derivations containing no ω_c transitions, thus $\langle G, A, S, B, T\rangle_n = \langle H, A, S', B', T'\rangle_{n'}$ so it immediately follows that $\langle G \mathbin{+\!\!+} A', S, B, T\rangle_n = \langle H \mathbin{+\!\!+} A', S', B', T'\rangle_{n'}$ which is a trivial ω_r derivation.

Induction Step: Suppose all ω_c derivations D_i consisting of i transitions (and where all subcomputations are refined), are refined. We show that the same is true for similar derivations of $i+1$ transitions.

Let $\sigma_i^c = \langle G_i, A, S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i. Let D_i^r be the corresponding ω_r derivation that exists because D_i is refined, and let $\sigma_i^r = \langle G_i \mathbin{+\!\!+} A', S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i^r. We consider all possible transitions between σ_i^c and σ_{i+1}^c in constructing a D_{i+1}, then show how to construct an ω_r

derivation D^r_{i+1} which satisfies the definition of *refined*.

CASE **Solve**: Then $G_i = c$ where c is a built-in constraint. Hence

$$\sigma^c_{i+1} = \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}$$

where S_{i+1}, B_{i+1}, T_{i+1} and n_{i+1} are given by the subcomputation

$$\langle wakeup\_policy(S_i, c, B_i), A, S_i, c \wedge B_i, T_i \rangle_{n_i} \rightarrowtail^* \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}} \quad (5.5)$$

if $\mathcal{D} \models_{\mathcal{S}} \bar{\exists}_{\emptyset}(c \wedge B)$, otherwise $S_{i+1} = S_i$, $B_{i+1} = c \wedge B_i$, $T_{i+1} = T_i$ and $n_{i+1} = n_i$ if $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset}(c \wedge B)$. By assumption, subcomputation (5.5) is refined, hence

$$\langle wakeup\_policy(S_i, c, B_i) +\!\!+ A', S_i, c \wedge B_i, T_i \rangle_{n_i} \rightarrowtail^* \langle A', S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}} \quad (5.6)$$

Notice the last state is in the appropriate form for σ^r_{i+1}.

Let σ^r be the result of applying ω_r **Solve** to σ^r_i, then

$$\sigma^r = \langle wakeup\_policy(S_i, c, B_i) +\!\!+ A', S_i, B_i, T_i \rangle_{n_i}$$

Furthermore,

$$D^r_{i+1} = D^r_i \rightarrowtail_{solve} \sigma_r \rightarrowtail^* \sigma^r_{i+1}$$

by derivation (5.6). Hence D_{i+1} is also refined.

CASE **Activate**: Then $G_i = c$ where c is a (non-numbered) CHR constraint. Hence

$$\sigma^c_{i+1} = \langle c\#n_i : 1, A, \{c\#n\} \uplus S_i, B_i, T_i \rangle_{(n_i+1)}$$

and

$$\sigma^r_{i+1} = \langle [c\#n_i : 1 | A'], \{c\#n\} \uplus S_i, B_i, T_i \rangle_{(n_i+1)}$$

after applying ω_r **Activate** to σ^r_i. Hence D_{i+1} is also refined.

CASE **Reactivate**: Then $G_i = c\#j$ a numbered CHR constraint. Hence

$$\sigma^c_{i+1} = \langle c\#j : 1, A, S_i, B_i, T_i \rangle_{n_i}$$

and

$$\sigma^r_{i+1} = \langle [c\#j : 1 | A'], S_i, B_i, T_i \rangle_{n_i}$$

after applying ω_r **Reactivate** to σ^r_i. Hence D_{i+1} is also refined.

CASE **Drop**: Then $G_i = c\#j : k$ where there is no such occurrence k of predicate c in P. Hence

$$\sigma^c_{i+1} = \langle \Box, A, S_i, B_i, T_i \rangle_{n_i}$$

and

$$\sigma^r_{i+1} = \langle A', S_i, B_i, T_i \rangle_{n_i}$$

after applying ω_r **Drop** to σ^r_i. Hence D_{i+1} is also refined.

CASE **Simplify**: Then $G_i = c\#j : k$ and the conditions for ω_c **Simplify** hold. Assuming that the rule can fire, we have that

$$\sigma^c_{i+1} = \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}$$

where the appropriate fields are given by the following subcomputation.

$$\begin{array}{c} \langle \theta(C), [c\#j : k|A], S_i - H_2 - H_3 - \{c\#j\}, B_i, T'_i \rangle_{n_i} \\ \rightarrowtail^* \langle \Box, [c\#j : k|A], S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}} \end{array} \tag{5.7}$$

Where C, H_2, H_3, θ and T_{i+1} are defined by ω_c **Simplify** (see Definition 23). By assumption, subcomputation (5.7) is refined, hence

$$\langle \theta(C) +\!\!+ A', S_i - H_2 - H_3 - \{c\#j\}, B_i, T'_i \rangle_{n_i} \rightarrowtail^* \langle A', S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}} \tag{5.8}$$

Notice the last state is in the appropriate form for σ^r_{i+1}.

Let σ^r be the result of applying ω_r **Simplify** to σ^r_i, then

$$\sigma^r = \langle \theta(C) +\!\!+ A', S_i - H_2 - H_3 - \{c\#j\}, B_i, T'_i \rangle_{n_i}$$

Furthermore,

$$D^r_{i+1} = D^r_i \rightarrowtail_{simplify} \sigma_r \rightarrowtail^* \sigma^r_{i+1}$$

by derivation (5.8). Hence D_{i+1} is also refined.

The other case is when the rule is not applicable. Then

$$\sigma^c_{i+1} = \langle c\#j : k+1, A, S_i, B_i, T_i \rangle_{n_i}$$

and

$$\sigma^r_{i+1} = \langle [c\#j : k+1|A'], S_i, B_i, T_i \rangle_{n_i}$$

after applying ω_r **Default**. Hence D_{i+1} is also refined.

CASE **Propagate**: Then $G_i = c\#j : k$ and the conditions for ω_c **Propagate** hold. We have that

$$\sigma^c_{i+1} = \langle G, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}$$

where the appropriate fields are given by the following series of subcomputations.

$$\begin{array}{c} \langle \theta(C), [c\#j : k|A], S_{l-1} - H_{3l}, B_{l-1}, T_{l-1} \rangle_{n_{l-1}} \rightarrowtail^* \\ \langle \Box, [c\#j : k|A], S_l, B_l, T_l \rangle_{n_l} \end{array} \tag{5.9}$$

Where all of the appropriate fields are defined by ω_c **Propagate** (see Definition 23). By assumption, each subcomputation (5.9) is refined, hence

$$\begin{array}{c} \langle \theta(C) +\!\!+ [c\#j : k|A'], S_{l-1} - H_{3l}, B_{l-1}, T_{l-1} \rangle_{n_{l-1}} \rightarrowtail^* \\ \langle [c\#j : k|A'], S_l, B_l, T_l \rangle_{n_l} \end{array} \tag{5.10}$$

Suppose that there are k sub-derivations, then we can construct ω_r derivation D^r_{i+1} as follows

$$D^r_{i+1} = D^r_i \rightarrowtail_{propagate} D_{l=1} \rightarrowtail_{propagate} \ldots \rightarrowtail_{propagate} D_{l=k} \rightarrowtail_{default} \sigma^r_{i+1}$$

where each sub-derivation D_l is constructed by applying **Propagate** (whose applicability is easily verified) to the last state in D_{l-1} (or σ_i^r if $l = 1$), and then the proceeding derivation defined by (5.10). Hence D_{i+1} is also refined.
CASE **Goal**: Then $G_i = [c|C]$ is a sequence of constraints. Hence

$$\sigma_{i+1}^c = \langle C, A, S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$$

where S_{i+1}, B_{i+1}, T_{i+1} and n_{i+1} are given by the subcomputation

$$\langle c, A, S_i, B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle \square, A, S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} \tag{5.11}$$

By assumption, subcomputation (5.11) is refined, hence

$$\langle [c|C] +\!\!+ A', S_i, B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle C +\!\!+ A', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} \tag{5.12}$$

Notice the last state is in the appropriate form for σ_{i+1}^r.

Now $\sigma_i^r = \langle [c|C] +\!\!+ A', S_i, B_i, T_i\rangle_{n_i}$, so we can now construct D_{i+1}^r as follows

$$D_{i+1}^r = D_i^r \rightarrowtail^* \sigma_{i+1}^r$$

by derivation (5.12). Hence D_{i+1} is also refined.

Therefore ω_c derivations D where all subcomputations are refined, are refined. □

The next Lemma is the same as the previous one except some assumptions are removed.

Lemma 6 *All ω_c derivations D, of the form*

$$\langle G, A, S, B, T\rangle_n \rightarrowtail^* \langle H, A, S', B', T'\rangle_{n'}$$

are refined.

Proof. By induction over subcomputations.

Base Case: D_0 has no subcomputations, then by Lemma 5 all possible D_0 are refined.

Induction Step: Suppose that for all ω_c derivations D_i where the maximum depth of nested subcomputations of D_i is less than or equal to i, are refined. Then we show that similarly defined D_{i+1}, which has at least one D_i subcomputation, is also refined. This also immediately follows from Lemma 5, since all subcomputations for D_{i+1} are refined by construction.

Therefore all ω_c derivations D, of the form

$$\langle G, A, S, B, T\rangle_n \rightarrowtail^* \langle H, A, S', B', T'\rangle_{n'}$$

are refined. □

5.3.2 From Refined to Call-based

In this subsection we show that some refined derivations can be associated with equivalent call-based derivations.

First we define the notion of a *subcomputation* under the refined semantics. These will correspond with call-based subcomputations.

Definition 26 (Refined Subcomputation) *Let $\sigma = \langle C \mathbin{+\!\!+} A, S, B, T\rangle_n$ be the result after applying a **Solve**, **Simplify** or **Propagate** transition under ω_r to some refined execution state. We interpret C as the result of the wakeup policy in the case of **Solve**, or C as the (renamed) body of the rule otherwise. We define a* subcomputation *to be the derivation*

$$\sigma \rightarrowtail^* \langle A, S', B', T'\rangle_{n'} = \sigma'$$

where σ' is the first state in this form (i.e. the first state with just A as the execution stack). □

Now we given the definition of a *call-based* ω_r derivation. This is analogous to Definition 25, which defined a *refined* ω_c derivation.

Definition 27 (Call-based) *We say a ω_r derivation*

$$\langle G \mathbin{+\!\!+} A, S, B, T\rangle_n \rightarrowtail^* \langle A, S', B', T'\rangle_{n'}$$

is call-based *if for all A' the following are ω_c derivations (assuming the same wakeup policy is used):*

$$\langle G, A', S, B, T\rangle_n \rightarrowtail^* \langle \Box, A', S', B', T'\rangle_{n'}$$

□

The following lemma shows that a derivation which executes a single constraint is called-based, assuming that all subcomputations are also call-based.

Lemma 7 *All derivations of the form $\langle [C|A], S, B, T\rangle_n \rightarrowtail^* \langle [C'|A], S', B', T'\rangle_{n'}$ (where C and C' is a built-in, CHR, numbered or active constraint) are call-based if*

1. *the derivation contains no **Solve**, **Drop** or **Simplify** transitions, except in subcomputations; and*
2. *all subcomputations are call-based.*

Proof. By induction.

Base Case: Derivations containing no ω_r transitions, thus $\langle [C|A], S, B, T\rangle_n = \langle [C'|A], S', B', T'\rangle_{n'}$ so it immediately follows that $\langle C, A', S, B, T\rangle_n = \langle C', A', S', B', T'\rangle_{n'}$ which is a trivial ω_c derivation.

Induction Step: Suppose all ω_r derivations D_i (which satisfy our conditions), are call-based. Here i is the number of transitions in the corresponding ω_c derivation for D_i. We show that the same is true for similar derivations D_{i+1} transitions constructed from some D_i.

Let $\sigma_i^r = \langle [C_i|A], S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i. Let D_i^c be the corresponding ω_c derivation that exists because D_i is call-based, and let $\sigma_i^c = \langle C_i, A', S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i^c. We consider all possible transitions applicable to σ_i^r, then show how to construct a ω_r derivation D_{i+1} and a corresponding ω_c derivation D_{i+1}^c which satisfies the definition of *call-based*. Let σ_{i+1}^r be the last state in the resulting D_{i+1}.

CASE **Activate**: Then $C_i = c$ where c is a (non-numbered) CHR constraint. Hence

$$\sigma_{i+1}^r = \langle [c\#n_i : 1|A], \{c\#n_i\} \uplus S_i, B_i, T_i\rangle_{(n_i+1)}$$

and

$$\sigma_{i+1}^c = \langle c\#n_i : 1, A', \{c\#n_i\} \uplus S_i, B_i, T_i\rangle_{(n_i+1)}$$

after applying ω_c **Activate**. Hence D_{i+1} is also call-based.

CASE **Reactivate**: Then $C = c\#j$ a numbered CHR constraint. Hence

$$\sigma_{i+1}^r = \langle [c\#j : 1|A], S_i, B_i, T_i\rangle_{n_i}$$

and

$$\sigma_{i+1}^c = \langle c\#n_i : 1, A', S_i, B_i, T_i\rangle_{n_i}$$

after applying ω_c **Reactivate**. Hence D_{i+1} is also call-based.

CASE **Propagate**: Then $C_i = c\#j : k$ and the conditions for ω_r **Propagate** hold.

Consider the derivation D_{i+1} constructed by exhaustively applying ω_r **Propagate** to given the rule and active constraint until no further application is possible.

$$D_{i+1} = D_i^r \rightarrowtail_{propgate} D_{l=1} \rightarrowtail_{propgate} \ldots \rightarrowtail_{propgate} D_{l=k} \rightarrowtail_{default} \sigma_{i+1}^r$$

where each subcomputation D_l is of the from

$$\langle \theta(C) +\!\!+ [c\#j : k|A], S_{l-1} - H_3, B_{l-1}, T'_{l-1}\rangle_{n_{l-1}} \rightarrowtail^* \langle [c\#j : k|A], S_l, B_l, T_l\rangle_{n_l}$$

where C, H_3, θ and T_i' are defined by ω_r **Propagate**. Here $S_0 = S_i$, $B_0 = B_i$, $T_0 = T_i$, $n_0 = n_i$ from σ_i^r. After the k^{th} subcomputation and the **Default** transition, the resulting state σ_{i+1}^r is

$$\langle [c\#j : k+1|A], S_k, B_k, T_k\rangle_{n_k}$$

By assumption, each D_l is call-based, hence the following are ω_c derivations.

$$\langle \theta(C), [c\#j : k|A'], S_{l-1} - H_3, B_{l-1}, T'_{l-1}\rangle_{n_{l-1}} \rightarrowtail^* \langle \Box, [c\#j : k|A'], S_l, B_l, T_l\rangle_{n_l}$$

The resulting state after executing each of these (call-based) subcomputations is therefore $\langle \Box, [c\#j : k|A'], S_k, B_k, T_k\rangle_{n_k}$.

We observe that we can apply ω_c **Propagate** to σ_i^c and arrive at the state

$$\sigma_{i+1}^c = \langle c\#j : k+1, A', S_k, B_k, T_k\rangle_{n_k}$$

by using the sequence of subcomputations defined above. Hence D_{i+1} is also call-based.
CASE **Default**: Then $C = c\#j : k$ and no other transitions is applicable to σ_i^r. Hence

$$\sigma_{i+1}^r = \langle [c\#j : k+1|A], S_i, B_i, T_i\rangle_{n_i}$$

and

$$\sigma_{i+1}^c = \langle c\#j : k+1, A', S_i, B_i, T_i\rangle_{n_i}$$

after applying ω_c **Simplify** or **Propagate** (depending on whether occurrence k deletes the active constraint or not). Hence D_{i+1} is also refined.

Therefore ω_c derivations D which satisfy our conditions are call-based. □

The following is similar to the previous lemma.

Lemma 8 *All derivations D of the form*

$$\langle [C|A], S, B, T\rangle_n \rightarrowtail^* \langle A, S', B', T'\rangle_{n'} = \sigma_f$$

(where C can be a built-in, CHR, numbered or active constraint) are call-based if

1. *σ_f is the only state in D of that form; and*
2. *and all subcomputations are call-based.*

Proof. Direct proof. We can divide up D as follows.

$$\sigma = \langle [C|A], S, B, T\rangle_n \rightarrowtail^* \langle [C'|A], S'', B'', T''\rangle_{n''} = \sigma' \rightarrowtail_{(solve \vee drop \vee simplify)} \rightarrowtail^* \sigma_f$$

The first part of the derivation $\sigma \rightarrowtail^* \sigma'$ satisfies the conditions for Lemma 7, and hence is call-based. Therefore

$$\langle C, A', S, B, T\rangle_n \rightarrowtail^* \langle C', A', S'', B'', T''\rangle_{n''} = \sigma'^c$$

under the ω_c semantics.

We consider all possible transitions applicable to σ' (i.e. **Solve**, **Drop** or **Simplify**), and show that D must be call-based.
CASE **Solve**: Then $C' = c$ where c is a built-in constraint. Hence

$$\sigma' \rightarrowtail_{solve} \langle wakeup\_policy(S'', c, B'') +\!\!+ A, S'', c \wedge B'', T''\rangle_{n''}$$

By assumption, the subcomputation

$$\langle wakeup\_policy(S'', c, B'') +\!\!+ A, S'', c \wedge B'', T''\rangle_{n''} \rightarrowtail^* \langle A, S', B', T'\rangle_{n'}$$

is call-based, thus

$$\sigma'^c \rightarrowtail_{solve} \langle \Box, A', S', B', T' \rangle_{n'}$$

Hence D is call-based.
CASE **Drop**: Then $C' = c\#j : k$ where there is no such occurrence k of predicate c in P. Hence

$$\sigma' \rightarrowtail_{drop} \langle A, S'', B'', T'' \rangle_{n''} = \sigma_f$$

and

$$\sigma'^c \rightarrowtail \langle \Box, A', S'', B'', T'' \rangle_{n''}$$

after applying ω_c **Drop**. Hence D is call-based.
CASE **Simplify**: Then $C' = c\#j : k$ and the conditions for ω_r **Simplify** hold. Hence

$$\sigma' \rightarrowtail_{solve} \langle \theta(C) +\!\!+ A, S'' - H_2 - H_3 - \{c\#j\}, B'', T''' \rangle_{n'} = \sigma''$$

where C, H_2, H_3, θ and T''' are defined by ω_r **Simplify** (see Definition 11). By assumption, the subcomputation

$$\sigma'' \rightarrowtail^* \langle A, S', B', T' \rangle_{n'}$$

is call-based, thus

$$\sigma'^c \rightarrowtail_{simplify} \langle \Box, A', S', B', T' \rangle_{n'}$$

Hence D is call-based.

Therefore, all such derivations D are call-based. □

This lemma is similar (but more general) than the previous two.

Lemma 9 *All derivations D of the form*

$$\langle A +\!\!+ A', S, B, T \rangle_n \rightarrowtail^* \langle A', S', B', T' \rangle_{n'} = \sigma_f$$

(where A contains only built-in or CHR constraints) are call-based if

1. *σ_f is the only state in D of that form; and*
2. *all subcomputations are call-based.*

Proof. By induction.

Base case: Derivations containing no ω_r transitions, i.e. $A = []$, thus $\langle A +\!\!+ A', S, B, T \rangle_n = \langle A', S', B', T' \rangle_{n'}$ and so it immediately follows that $\langle A', A'', S, B, T \rangle_n = \langle A', A'', S', B', T' \rangle_{n'}$ which is a trivial ω_c derivation.

Induction step: Suppose that D_i is call-based for all derivations D_i of the following form

$$\langle A_i +\!\!+ A', S_i, B_i, T_i \rangle_{n_i} \rightarrowtail^* \langle A', S', B', T' \rangle_{n'}$$

where the length of A_i is i, and D_i satisfies the same conditions as D. We show the same is true for all similarly defined derivations D_{i+1}. Note $A_{i+1} = [C|A_i]$ for some C.

Let the first state in D_{i+1} be $\langle [C|A_i], S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$. Let D_i^c be the ω_c derivation of the form

$$\langle A_i, A'', S_i, B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle A', A'', S', B', T'\rangle_{n'}$$

that exists because D_i is call-based.

By Lemma 8 the following is also a call-based derivation:

$$\langle C, A'', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} \rightarrowtail^* \langle \Box, A'', S_i, B_i, T_i\rangle_{n_i} \tag{5.13}$$

Consider the call-based state

$$\sigma_{i+1}^c = \langle [C|A_i], A'', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$$

then

$$\sigma_{i+1}^c \rightarrowtail_{goal} \langle A_i, A'', S_i, B_i, T_i\rangle_{n_i}$$

by using the subcomputation in (5.13). Thus we have constructed a derivation D_{i+1}^c of the required form to show that D_{i+1} is call-based.

Therefore, all derivations D which satisfy our conditions are call-based. □

The final lemma is equivalent to the previous one, except the additional assumptions have been removed.

Lemma 10 *All ω_r derivations D, of the form*

$$\langle A +\!\!+ A', S, B, T\rangle_n \rightarrowtail^* \langle A', S', B', T'\rangle_{n'}$$

where A contain only built-in or CHR constraints, are call-based.

Proof. By induction over subcomputations.

Base Case: D_0 has no subcomputations, then by Lemma 9 all possible D_0 are call-based.

Induction Step: Suppose that for all ω_r derivations D_i where the maximum depth of nested subcomputations of D_i is less than or equal to i, are call-based. Then we show that similarly defined D_{i+1}, which has at least one D_i subcomputation, is also call-based. This also immediately follows from Lemma 9, since all subcomputations for D_{i+1} are call-based by construction.

Therefore all ω_r derivations D, of the form

$$\langle A +\!\!+ A', S, B, T\rangle_n \rightarrowtail^* \langle A', S', B', T'\rangle_{n'}$$

are call-based. □

5.3.3 Main Result

Theorem 4 (Equivalence of Semantics) *For an initial goal G;*

$$\langle G, \emptyset, true, \emptyset \rangle_1 \rightarrowtail^* \langle [], S, B, T \rangle_n$$

under the refined semantics iff

$$\langle G, [], \emptyset, true, \emptyset \rangle_1 \rightarrowtail^* \langle \Box, [], S, B, T \rangle_n$$

under the call-based semantics.

Proof. Immediately follows from Lemma 6 and Lemma 10. □

5.4 Abstract Interpretation Framework

In this section we present the generic abstract interpretation framework for CHRs. The framework for CHRs is based on an abstraction of the call-based operational semantics given in the Section 5.2. Instead of a concrete state, an abstract state is used. Similarly, abstract transition rules are used instead of concrete ones.

5.4.1 Abstract State

Every instance of the abstract interpretation framework should define a domain Σ_a of abstract states. The abstract domain Σ_a has to be a lattice with partial ordering $\preceq$, least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operations.

Furthermore an abstraction function α has to be defined from a concrete state σ, as defined in Section 5.2, to an abstract state s and a concretisation function γ from an abstract state to a set of concrete states.[2] It should be possible to determine from the abstract state whether it is a final state, i.e. with an empty goal $\Box$.

5.4.2 Abstract Transitions

The abstract domain must provide the following abstract operations corresponding to the transitions in the call based semantics: AbstractSolve, AbstractActivate, AbstractReactivate, AbstractDrop, AbstractSimplify, AbstractPropagate and AbstractGoal. These abstract operations are abstractions of the transition rules defined by the call-based operational semantics of CHR, as given in Section 5.2.

With the exception of AbstractSimplify, the abstract transitions are transitions of the form $\Sigma_a \rightarrowtail \Sigma_a$. In order for the abstract transition rules to be consistent abstractions of the concrete transition rules, we impose the connection depicted below:

[2] Typically γ is defined in terms of α, e.g. $\gamma(s) = \{\sigma | \alpha(\sigma) = s\}$.

$$\begin{array}{ccc} \sigma_1 & \rightarrowtail & \sigma_2 \\ {\scriptstyle\alpha}\downarrow & & \uparrow{\scriptstyle\gamma} \\ s_1 & \rightarrowtail & s_2 \end{array}$$

In other words, if $\sigma_1 \rightarrowtail \sigma_2$ then if $\alpha(\sigma_1) \rightarrowtail s$ under the abstract semantics, we have that $\sigma_2 \in \gamma(s)$.

The AbstractSimplify transition is of the form $\Sigma_a \rightarrowtail \Sigma_a \cup (\Sigma_a \times \Sigma_a)$, We denote $\Sigma_a \cup (\Sigma_a \times \Sigma_a)$ as answers which is defined as:

$$\mathsf{answers} ::= \mathtt{one}(\Sigma_a) \mid \mathtt{two}(\Sigma_a, \Sigma_a)$$

Its meaning is clear, there are either one or two possible resulting states. The following conditions must hold: for all concrete states σ_1 and σ_2 such that $\sigma_1 \rightarrowtail_{simplify} \sigma_2$ we have that

- if $\alpha(\sigma_1) \rightarrowtail_{AS} \mathtt{one}(s)$ then $\sigma_2 \in \gamma(s)$; and
- if $\alpha(\sigma_1) \rightarrowtail_{AS} \mathtt{two}(s_1, s_2)$ then $\sigma_2 \in \gamma(s_1)$ or $\sigma_2 \in \gamma(s_2)$.

The way multiple resulting states are combined by the framework is discussed below.

5.4.3 The Generic Abstract Semantics

Here we explain the generic semantics of the framework, based on the analysis-specific implementations of the abstract domain and abstract transition rules.

The concrete operational semantics specify that a program starts from an initial state and transition rules are applied until a final state is reached. In the following we describe what initial state is used by the framework and how the final state is obtained by applying abstract transition rules. In particular the issues of nondeterminism are discussed.

Generic initial state

For any CHR program, an infinite number of concrete initial states are possible, namely any $\langle G, [], \emptyset, \emptyset, \emptyset \rangle_1$ where G is any finite list of CHR and built-in constraints. This infinite number of initial states may lead to an infinite number of abstract states, depending on the definition of α.

One approach for handling an infinite number of initial states is to use the following fix-point computation. Say $\{c_i | 1 \leq i \leq n\}$ are all the possible distinct abstract CHR and built-in constraints, then starting from abstract state $s_0 = \alpha(\langle \Box, [], \emptyset, true, \emptyset \rangle_n)$, the final state s_f is s_k, where:

$$s_j = \bigsqcup \{ s_j^i | \mathsf{new\_goal}(s_{j-1}, c_i) \rightarrowtail^* s_j^i \wedge \mathsf{final}(s_j^i) \wedge 1 \leq i \leq n \}$$

for $j > 0$ and k is the smallest integer such that $s_k = s_{k+1}$. In the above formula, new_goal is the function that replaces the empty goal in a final abstract state with a new goal c_i, and final is a test that succeeds if the given state is a final state. This approach only works if the number of abstract built-in constraints is finite.

Transition application

The generic framework applies the abstract transitions on an initial state until a final state is reached. For most abstract states, the result of applying a transition is another single abstract state, so the framework's task is straightforward. The AbstractSimplify is an exception, as already mentioned in Section 5.4.2.

It is the framework's task to take the determinism into account and compute the appropriate results from the two alternate possibilities. Consider an abstract state s_0 where the AbstractSimplify transition applies. If $s_0 \rightarrowtail_{AS} \texttt{one}(s_1)$ then s_1 is the resulting state. If $s_0 \rightarrowtail \texttt{two}(s_1, s_2)$ then there are two possible results. In order to find a least upper bound we must extend the states to final states and then build the least upper bound. The framework then computes the following final state s_* for s_0:

$$s_* = \begin{cases} s_1 & \text{if } s_0 \rightarrowtail_{AS} \texttt{one}(s_1) \\ s_1^* \sqcup s_2^* & \text{if } s_0 \rightarrowtail_{AS} \texttt{two}(s_1, s_2) \\ & \text{and } s_1 \rightarrowtail^* s_1^*, s_2 \rightarrowtail^* s_2^* \end{cases}$$

with $\rightarrowtail_{AS}$ an application of the AbstractSimplify Rule.

Nondeterminism in the Simplify and Propagate transitions

While the above accounts for the nondeterminism in simplification matching caused by abstraction, it does no account for the inherent nondeterminism of these transitions in the concrete semantics.

Namely, for a simplification transition, if more than one combination of partner constraints are possible, the concrete semantics do not specify what particular combination is chosen. To account for this nondeterminism the formulation of the AbstractSimplify transition should capture all possible concrete transitions. In particular, if for concrete state σ there n different possible resulting states $\sigma_1, \ldots, \sigma_n$, then $\alpha(\sigma) \rightarrowtail_{AS} \texttt{one}(s)$ or $\alpha(\sigma) \rightarrowtail_{AS} \texttt{two}(s, \_)$ such that $\bigsqcup_{i=1}^{n} \alpha(\sigma_i) \preceq s$.

Similarly, for a propagation transition, multiple combination transitions are possible. In addition, for a propagation transition, multiple applications are possible in a sequence. However, the order of the sequence is not specified by the concrete semantics either. Hence, an abstract propagation transition has to capture all possible partner combinations and all possible sequences in which they are dealt with.

Nondeterminism in the Solve transition

The nondeterminism inherent in the concrete **Solve** transition lies in the order the woken up constraints (the ones chosen by the *wakeup_policy*) are executed: all possible orderings are allowed. An abstract domain has to provide an abstraction that takes into account all possible orderings.

First, the abstraction of the *wakeup_policy* itself must be considered. Usually the abstraction over approximates the set of constraints that are woken up. One

possible safe approximation is to wakeup all potentially non-fixed constraints. In HAL, we can use mode information to decide which constraints are always fixed/ground at runtime, i.e. when all arguments to the constraint have mode '`in`'. We will assume the existence of a test $\mathsf{is\_ground}(c)$, which succeeds if c must be ground. We define the default abstract *wakeup_policy* as

$$\mathsf{wakeup\_policy}(S) = \{c \mid c \in S \wedge \neg \mathsf{is\_ground}(c)\}$$

where S is the abstract store.[3]

If the abstract *wakeup_policy* wakes up a non-empty set of constraints during a AbstractSolve, then the order these constraints are executed must be considered. One approach would be, if the abstract domain permits, to compute the final state s_o for all $o \in O$ with O the set of all possible orderings and to combine these final states to a single final state s as follows: $s = \bigsqcup_{o \in O} s_o$. However, this requires sufficiently concrete information about the number of woken up constraints in the abstract domain. Typically the abstract domain cannot provide any quantitative bound on the number of woken up constraints. Hence an infinite number of orderings are possible: all possible permutations of constraint sequences of any integer length.

A possible finite approximation of this infinite number of possibilities is to perform a fix-point computation similar to the one used for the initial goal. Say $\{c_i | 1 \leq i \leq n\}$ are all the possible distinct abstract CHR constraints woken up by the abstract *wakeup_policy*, then starting from abstract state s_0, the final state s_f after waking up all constraints in any quantity is s_k, where:

$$s_j = \bigsqcup \{s_j^i | \mathsf{new\_goal}(s_{j-1}, c_i) \rightarrowtail^* s_j^i \wedge \mathsf{final}(s_j^i) \wedge 1 \leq i \leq n\}$$

for $j > 0$ and k is the smallest integer such that $s_k = s_{k+1}$. This generic approach is illustrated in the functional dependency analysis (see Section 5.6). Due to its generality, this approach may cause a huge loss of precision as well as an exponential number of intermediate states. Hence, in practice, better domain specific techniques should be studied.

5.5 Late Storage Analysis

In this section we illustrate the use of the abstract interpretation framework for CHR with a CHR-specific analysis: late storage. This is a useful analysis that drive some CHR optimisations, which are discussed in Chapter 7.

[3]This abstract *wakeup_policy* does not take into consideration the built-in constraint used in the AbstractSolve, nor the abstract built-in store. Of course, if the domain permits, a more accurate abstract *wakeup_policy* which takes these into consideration may be possible.

5.5.1 The Observation Property

Late storage analysis returns information about what CHR constraints are *observed* during a given subcomputation. We formally define the property of *observed* as follows.

Definition 28 (Observed) *An identified constraint $c\#i$ is* observed *in a subcomputation D if there exists a transition in D, or any subcomputation of D, of any of the following forms:*

1. *There exists a transition*

$$\langle c', A, S, B, T\rangle_n \rightarrowtail_{solve} \langle \Box, A, S', B', T'\rangle_{n'}$$

and $c\#i \in wakeup\_policy(S, c, B)$

2. *There exists a transition*

$$\langle c'\#i' : j, A, \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{simplify} \langle \Box, A, S', B', T'\rangle_{n'}$$

where H_1, H_2 and H_3 are the constraints from the store that match H'_1, H'_2 and H'_3 from the (renamed copy of the) rule (see Definition 23), and $c\#i \in H_1 \cup H_2 \cup H_3$.

3. *There exists a transition*

$$\langle c'\#i' : j, A, S, B, T\rangle_n \rightarrowtail_{propagate} \langle G, A, S, B, T\rangle_n$$

which has at least one subcomputation with H_{1l}, H_{2l} and H_{3l} from S_{l-1} (see Definition 23), and $c\#i \in H_{1l} \cup H_{2l} \cup H_{3l}$

□

Informally, a constraint in the constraint store is observed, if it is reactivated by a built-in constraint or if it serves as a matching partner to an active constraint.

Observation is an interesting property because if a constraint $c\#i$ remains *unobserved* during a derivation D, we can construct a new derivation D' by removing $c\#i$ from the CHR store of all states in D and subcomputations of D. Since $c\#i$ does not affect the applicability of any transitions (by definition), D' is a valid ω_c derivation.

One application for this information is program optimisation. If the compiler knows that any subcomputation for the first occurrence (of some constraint) will not observe the active constraint, then the compiler can delay inserting the constraint in the store to the next occurrence. This is opposed to the basic compilation, which always inserts active constraints in the top-level predicate. We refer to this as the *late storage optimisation*, which lends its name to the late storage analysis of this section. The advantage of late storage is that if the active constraint is deleted before it has been inserted into the store, we have saved the

cost of an insertion and deletion. The implementation of this optimisation will be discussed later in Section 7.3.2.

To correctly define the analysis of *observation* as an abstract interpretation we have to extend the call-based operational semantics to make this visible. We will only be interested in finding the *observed* occurrences of constraints in the activation stack.

Definition 29 *We denote an* observed *occurrence $c\#i : j$ by marking it with a star $*$, e.g. $c\#i : j^*$, and define*

$$\begin{array}{rcll} obs(c\#i : j) & = & c\#i : j^* & \\ obs(c\#i : j^*) & = & c\#i : j^* & \\ obs([], S) & = & [] & \\ obs([c\#i : j|G], S) & = & [obs(c\#i : j)|obs(G, S)] & c\#i \in S \\ obs([c\#i : j|G], S) & = & [c\#i : j|obs(G, S)] & c\#i \notin S \end{array}$$

□

We only need to redefine the **Solve**, **Simplify** and **Propagate** transitions slightly. We modify the activation stack to record which constraints have been observed by any of these transitions.

Definition 30 (Extended Call-based Operational Semantics)
1. Solve

$$\langle c, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \Box, A', S', B', T'\rangle_{n'}^{\mathcal{V}}$$

where c is a built-in constraint. If $\mathcal{D} \models_{\mathcal{S}} \neg\bar{\exists}_{\emptyset}(c \wedge B)$, then $S' = S$, $B' = c \wedge B$, $T' = T$, $n' = n$. Otherwise $\mathcal{D} \models_{\mathcal{S}} \bar{\exists}_{\emptyset}(c \wedge B)$, where

$$\langle S_1, obs(A, S_1), S, c \wedge B, T\rangle_n^{\mathcal{V}} \rightarrowtail^* \langle \Box, A', S', B', T'\rangle_{n'}^{\mathcal{V}}$$

and $S_1 = wakeup\_policy(S, c, B)$.
5. Simplify

$$\langle c\#i : j, A, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle \Box, A', S', B', T''\rangle_{n'}^{\mathcal{V}}$$

where

$$\langle \theta(C), obs(A, H_1 \cup H_2 \cup H_3), H_1 \uplus S, B, T'\rangle_n^{\mathcal{V}} \rightarrowtail^* \langle \Box, A', S', B', T''\rangle_{n'}^{\mathcal{V}}$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

and there exists matching substitution θ is such that

$$\begin{cases} c = \theta(d_j) \\ cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ cons(H_3) = \theta(H_3') \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

In the intermediate transition sequence $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.

If no such matching substitution θ exists then

$$\langle c\#i : j, A, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle c\#i : j+1, A, S, B, T\rangle_n^{\mathcal{V}}$$

6. Propagate

$$\langle c\#i : j, A, \{c\#i\} \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle G, A_k, S_k, B_k, T_k\rangle_{n_k}^{\mathcal{V}}$$

where the j^{th} occurrence of the CHR predicate of c in a rule in P is

$$r \ @ \ H'_1, d_j, H'_2 \setminus H'_3 \Longleftrightarrow g \mid C$$

Let $A_0 = A$, $S_0 = S \uplus \{c\#i\}, B_0 = B, T_0 = T, n_0 = n$. Now assume, for $1 \leq l \leq k$ and $k \geq 0$, the series of transitions

$$\begin{array}{c}\langle \theta_l(C_l), [c\#i : j | A'_{l-1}], H_{1l} \uplus \{c\#i\} \uplus H_{2l} \uplus S'_{l-1}, B_{l-1}, T_{l-1} \cup \{t_l\}\rangle_{n_{l-1}}^{\mathcal{V}} \\ \rightarrowtail^* \langle \Box, [\_ | A_l], S_l, B_l, T_l\rangle_{n_l}^{\mathcal{V}}\end{array}$$

where $A'_{l-1} = obs(A_{l-1}, H_{1l} \cup H_{2l} \cup H_{3l})$, $\{c\#i\} \uplus H_{1l} \uplus H_{2l} \uplus H_{3l} \uplus S'_{l-1} = S_{l-1}$ and there exists a matching substitution θ_l, and a renamed copy of rule r

$$r \ @ \ H'_{1l}, d_{jl}, H'_{2l} \setminus H'_{3l} \Longleftrightarrow g_l \mid C_l$$

such that

$$\begin{cases} c = \theta_l(d_{jl}) \\ cons(H_{1l}) = \theta_l(H'_{1l}) \\ cons(H_{2l}) = \theta_l(H'_{2l}) \\ cons(H_{3l}) = \theta_l(H'_{3l}) \\ \mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta_l \wedge g_l) \\ t_l = id(H_{1l}) +\!\!+ [i] +\!\!+ id(H_{2l}) +\!\!+ id(H_{3l}) +\!\!+ [r] \notin T_{l-1} \end{cases}$$

Furthermore, for $k+1$ no such transition is possible.

The resulting goal G is either $G = \Box$ if $\mathcal{D} \models_{\mathcal{S}} \bar{\exists}_{\emptyset}(\neg B_k)$ (i.e. failure occurred) or $G = c\#i : j+1$ otherwise.

2. Activate, **3. Reactivate**, **4. Drop** and **7. Goal**
Same as Definition 23. □

Example 49 *When examining the derivation shown in Example 48 the altered versions of the transitions above make one change. After the* ***Simplify*** *transition in the derivation for s, the p in the store is observed, so the new state is*

$$\rightarrowtail_{Si} \quad \langle \Box, [p\#1 : 4^*], \{q\#2\}\rangle_4$$

□

5.5.2 Abstract Domain

We formally define the abstraction function α_{ls} for late storage analysis as follows.

Definition 31 *Let c be a built-in constraint and p a CHR constraint, and S a set or multiset of CHR constraints, then*

$$\begin{aligned}
\alpha_{ls}(c) &= \texttt{builtin} \quad (c \text{ built-in}) \\
\alpha_{ls}(p(t_1,\ldots,t_n)) &= p \\
\alpha_{ls}(p(t_1,\ldots,t_n)\#i) &= p \\
\alpha_{ls}(p(t_1,\ldots,t_n)\#i:j) &= p{:}j \\
\alpha_{ls}([]) &= [] \\
\alpha_{ls}([c|G]) &= [\alpha_{ls}(c)|\alpha_{ls}(G)] \\
\alpha_{ls}(S) &= \{\alpha_{ls}(c)|c \in S\} \quad (S \text{ set}) \\
\alpha_{ls}(\langle G, A, _, _, _\rangle^{_}) &= \langle \alpha_{ls}(G), \mathit{unobserved}(A)\rangle
\end{aligned}$$

where $\mathit{unobserved}(A) = A - \mathit{observed}(A)$*, and* $\mathit{observed}(A)$ *is defined as*

$$\mathit{observed}(A) = \{p \mid p(t_1,\ldots,t_n)\#i : j^* \in A\}$$

□

The abstract state used for this analysis is rather simple. We abstract CHR constraints by their predicate names, and built-in constraints as simply the special predicate name `builtin`.[4] We abstract non-active CHR constraints by keeping the predicate. We abstract active CHR constraints by removing the identity number, but we still keep the occurrence number. We eliminate observed constraints from the execution stack using the auxiliary function unobserved.

The partial ordering on states is $\langle G, A\rangle \preceq_{ls} \langle G', A'\rangle$ iff $G = G'$ and $A' \subseteq A$. Clearly the abstract domain forms a lattice with the ordering relation $\preceq_{ls}$. The least upper bound operator $\sqcup_{ls}$ can be defined as follows:

$$\langle G, A_1\rangle \sqcup_{ls} \langle G, A_2\rangle = \langle G, (A_1 \cap A_2)\rangle$$

The concretisation function γ_{ls} is defined as $\gamma_{ls}(s) = \{\sigma \mid \alpha_{ls}(\sigma) \preceq_{ls} s\}$. Note that the analysis is imprecise, i.e. it only tracks which constraints are *possibly observed* on the constraint predicate level.

5.5.3 Abstract Transitions

Each abstract operation must provide two things:
(a) whether it is applicable at the current state s_0, and
(b) the resulting state afterwards s.

[4] We assume that no CHR constraint uses the special predicate name `builtin`.

Definition 32 (Abstract Transitions for Late Storage)
1. AbstractSolve

$$s_0 = \langle \texttt{builtin}, A \rangle$$

Let $S_1 = \mathsf{wakeup\_policy}(S)$, where S is the set of all possible abstract CHR constraints (which represents the weakest possible abstraction for the CHR store). Then

$$\begin{array}{ll} A_0 & = A - S_1 \\ s_j & = \langle \Box, A_j \rangle = \sqcup_{ls}\{s_j^i \mid \langle p_i, A_{j-1} \rangle \rightarrowtail^* s_j^i \wedge \mathsf{final}(s_j^i) \wedge 1 \leq i \leq n\}, j \geq 1 \end{array}$$

where p_i are predicates of all potentially nonground constraints, i.e. those that satisfy $\mathsf{is\_ground}(p_i)$. Then

$$s = \langle \Box, A_k \rangle$$

2. AbstractActivate and **3.** AbstractReactivate

$$s_0 = \langle c, A \rangle \rightarrowtail \langle c{:}1, A \rangle = s$$

Applicable if c is a CHR constraint (but not active).
4. AbstractDrop

$$s_0 = \langle c{:}j, A \rangle \rightarrowtail \langle \Box, A \rangle = s$$

Applicable if no occurrence j exists for CHR predicate c.
5. AbstractSimplify

$$s_0 = \langle c{:}j, A_0 \rangle$$

Applicable if occurrence j is a simplification occurrence

$$r \ @ \ H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

Let $O = \alpha_{ls}(H_1' \cup H_2' \cup H_3')$ and let $A_1 = A_0 - O$. Assume that

$$\langle \alpha_{ls}(C), A_1 \rangle \rightarrowtail^* \langle \Box, A_2 \rangle$$

Then $s_1 = \langle \Box, A_2 \rangle$.

We consider the following two cases for deriving the resulting state s.

- If rule r is an unconditional simplification rule (i.e. the guard is *true*) of the form

  $$p(x_1, ..., x_n)_j \iff C$$

  Rule application only fails when the active constraint is not in the constraint store, this leads to a state $\langle \Box, A_0 \rangle$ which when lubbed with s_1 gives s_1. Hence $s = \texttt{one}(s_1)$.

- Otherwise

  $$s = \texttt{two}(s_1, \langle p{:}(j+1), A_0 \rangle)$$

6. AbstractPropagate

$$s_0 = \langle c\!:\!j, A_0 \rangle$$

Applicable if occurrence j is a propagation occurrence

$$r \ @ \ H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$$

Let $O = \alpha_{ls}(H'_1 \cup H'_2 \cup H'_3)$, $A_1 = A_0 - O$. Let $A_2 = A_1 \cup \{\alpha_{ls}(c)\}$. Assume

$$\langle \alpha_{ls}(C), A_2 \rangle \rightarrowtail^* \langle \Box, A_3 \rangle$$

Let $A_4 = A_3 \setminus (\{\alpha_{ls}(c)\} \setminus A_1)$, removing $\alpha_{ls}(c)$ from the execution stack if it was not present initially. Then the result of the rule is

$$s = \langle c\!:\!j+1, A_4 \rangle$$

Note that the active constraint c may have been observed while executing C iff $c \notin A_3$.

Note here we treat the rule as if it always could have fired. This is clearly safe.

7. AbstractGoal

$$s_0 = \langle [c|G], A \rangle \rightarrowtail \langle G, A' \rangle = s$$

where

$$\langle c, A \rangle \rightarrowtail^* \langle \Box, A' \rangle$$

□

5.5.4 Example Analysis

Consider the execution of the goal `p` with respect to the following (numbered) CHR program

```
p1 ==> q.
p2, t1 <=> r.
p3, r1 ==> true.
p4 ==> s.
p5, s1 <=> true.
```

The example derivation is shown in Figure 5.2. For simplification rules we show the states s_1 and s_2 in the answer $\texttt{two}(s_1, s_2)$ after lines labelled first and second, and then give the two derivations that lead to the lub.

Note that we observe the p only in the derivation for s hence we can safely delay storage of p until just before the execution of this body.

$$
\begin{array}{ll}
 & \langle p, \emptyset \rangle \\
\rightarrowtail_{AA} & \langle p{:}1, \emptyset \rangle \\
\rightarrowtail_{AP} & \langle p{:}2, \emptyset \rangle \qquad \langle q, \{p\} \rangle \rightarrowtail^* \langle \Box, \{p\} \rangle \\
 & \multicolumn{2}{l}{\text{---------- first}} \\
 & \rightarrowtail_{ASi} \quad \langle \Box, \emptyset \rangle \qquad \langle r, \emptyset \rangle \rightarrowtail^* \langle \Box, \emptyset \rangle \\
 & \multicolumn{2}{l}{\text{---------- second}} \\
 & \rightarrowtail_{\neg ASi} \quad \langle p{:}3, \emptyset \rangle \\
 & \rightarrowtail_{AP} \quad \langle p{:}4, \emptyset \rangle \qquad \langle \Box, \{p\} \rangle \rightarrowtail^* \langle \Box, \{p\} \rangle \\
 & \rightarrowtail_{AP} \quad \langle p{:}5, \emptyset \rangle \qquad \langle s, \{p\} \rangle \rightarrowtail^* \langle \Box, \emptyset \rangle \\
 & \qquad \multicolumn{2}{l}{\text{---------- first}} \\
 & \qquad \rightarrowtail_{ASi} \quad \langle \Box, \emptyset \rangle \qquad \langle \Box, \emptyset \rangle \rightarrowtail^* \langle \Box, \emptyset \rangle \\
 & \qquad \multicolumn{2}{l}{\text{---------- second}} \\
 & \qquad \rightarrowtail_{\neg ASi} \langle p{:}6, \emptyset \rangle \\
 & \qquad \rightarrowtail_{ADp} \langle \Box, \emptyset \rangle \\
 & \qquad \multicolumn{2}{l}{\text{---------- lub}} \\
 & \rightarrowtail_{\sqcup} \quad \langle \Box, \emptyset \rangle \\
 & \multicolumn{2}{l}{\text{---------- lub}} \\
\rightarrowtail_{\sqcup} & \langle \Box, \emptyset \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \langle q, \{p\} \rangle \\
\rightarrowtail_{AA} & \langle q{:}1, \{p\} \rangle \\
\rightarrowtail_{ADp} & \langle \Box, \{p\} \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \langle r, \emptyset \rangle \\
\rightarrowtail_{AA} & \langle r{:}1, \emptyset \rangle \\
\rightarrowtail_{AP} & \langle r{:}2, \emptyset \rangle \qquad \langle \Box, \{r{:}1\} \rangle \rightarrowtail^* \langle \Box, \{r{:}1\} \rangle \\
\rightarrowtail_{ADp} & \langle \Box, \emptyset \rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \langle s, \{p\} \rangle \\
\rightarrowtail_{AA} & \langle s{:}1, \{p\} \rangle \\
 & \multicolumn{2}{l}{\text{---------- first}} \\
 & \rightarrowtail_{ASi} \quad \langle \Box, \emptyset \rangle \qquad \langle \Box, \emptyset \rangle \rightarrowtail^* \langle \Box, \emptyset \rangle \\
 & \multicolumn{2}{l}{\text{---------- second}} \\
 & \rightarrowtail_{\neg ASi} \quad \langle s{:}2, \{p\} \rangle \\
 & \rightarrowtail_{ADp} \quad \langle \Box, \{p\} \rangle \\
 & \multicolumn{2}{l}{\text{---------- lub}} \\
\rightarrowtail_{\sqcup} & \langle \Box, \emptyset \rangle
\end{array}
$$

Figure 5.2: Example abstract derivation for late storage analysis

5.6 Functional Dependencies

5.6.1 The Functional Dependency Property

A functional dependency is a relationship between the arguments of a CHR constraint. The notation we use for functional dependencies is

$$p(x_1, ..., x_n) :: \{x_{i_0}, ..., x_{i_l}\} \rightsquigarrow \{x_{j_0}, ..., x_{j_m}\}$$

which indicates that the arguments $\{x_{i_0}, ..., x_{i_l}\}$ functionally determine the value of arguments $\{x_{j_0}, ..., x_{j_m}\}$, where both $\{x_{i_0}, ..., x_{i_l}\}$ and $\{x_{j_0}, ..., x_{j_m}\}$ are subsets of $\{x_1, ..., x_n\}$. We sometimes refer the domain $\{x_{i_0}, ..., x_{i_l}\}$ as the *key* for the functional dependency. Several optimisations discussed later in this thesis rely on functional dependencies, so an accurate analysis is important.

The key to detecting functional dependencies is the following utility function, which counts the number of constraints satisfying a particular form in the given CHR store.

Definition 33 *Given a functor/arity of a constraint p/n, a set of positive integers $\{i_0, ..., i_j\}$, CHR store S and a built-in store B, we define function $\mathsf{count}(p, n, \{i_0, ..., i_j\}, S, B)$ to be the following. Let $S' \subseteq S$ be the maximal (in size) subset of S such that all $p(x_1, ..., x_n)\#i \in S'$ and $p(y_1, ..., y_n)\#i' \in S'$ satisfy*

$$\mathcal{D} \models_{\mathcal{S}} B \rightarrow (x_{i_0} = y_{i_0} \wedge ... \wedge x_{i_j} = y_{i_j})$$

Then $\mathsf{count}(p, n, \{i_0, ..., i_j\}, S, B) = |S'|$. □

Example 50 *For example, given the following CHR store*

$$S = \{p(1, 2)\#1, p(1, 3)\#2, p(1, 4)\#3\}$$

then $\mathsf{count}(p, 2, \{1\}, S, true) = 3$ since there is at most three constraints which share the same first argument. Similarly, $\mathsf{count}(p, 2, \{2\}, S, true) = 1$ since there is at most one constraint which share the same second argument. □

We can formally define a functional dependency in terms of the count function as follows.

Definition 34 (Set Semantic Functional Dependency) *Given a CHR store S, and built-in store B, we say constraint p/n has a* set semantic functional dependency

$$p(x_1, ..., x_n) :: \{x_{i_0}, ..., x_{i_j}\} \rightsquigarrow \{x_1, ..., x_n\}$$

if $\mathsf{count}(p, n, \{i_0, ..., i_j\}, S, B) \leq 1$. □

Example 51 *Consider the CHR store S from Example 50. Then the set semantic functional dependency $p(x, y) :: \{y\} \rightsquigarrow \{x, y\}$ holds since each possible value*

of y is associated with at most one value of $\{x, y\}$ in the store. For example, $y = 3$ is only associated with $\{1, 3\}$ from the constraint $p(1, 3)\#2$, etc.

On the other hand, the set semantic functional dependency $p(x, y) :: \{x\} \rightsquigarrow \{x, y\}$ does not hold since there exists a value for x, namely $x = 1$, which is associated with multiple values for $\{x, y\}$, i.e., $\{1, 2\}$, $\{1, 3\}$, etc. □

This definition is only concerned with *full* functional dependencies, i.e. the domain of the functional dependency determines all other arguments. A more general functional dependency only needs to determine some subset of all arguments greater than the domain, e.g., $p(x, y, z) :: \{x\} \rightsquigarrow \{x, y\}$ is a non-full functional dependency. Currently, we only analyse for full functional dependencies. Some optimisations, such as indexing, require full functional dependencies.

Also, our definition differs from the usual mathematical definition of a functional dependency. Strictly speaking, a functional dependency is a relationship between arguments of constraints, namely what arguments determine the values of other arguments. In a *set semantic* functional dependency, we require both a traditional functional dependency, and the requirement that there is at most one copy of a constraint with the same key in the CHR store. For example, the CHR store

$$\{p(1, 2)\#1, p(1, 2)\#2\}$$

has a functional dependency between the first and second arguments of the `p` constraint, because given the value to the first argument we can determine the value of the second. However, there is no set semantic functional dependency, because two constraints with the same first argument simultaneously appear in the store at once. Set semantic functional dependencies are stronger than the traditional functional dependencies. All optimisations discussed later in this thesis specifically rely on set semantic functional dependencies, hence the distinction. For brevity, we will often refer "set semantic functional dependencies" as simply "functional dependencies" from now on. We will also occasionally use the term *set semantic* to indicate that only one copy of the constraint is present in the store at one time. This is equivalent to the set semantic function dependency $p(\bar{x}) :: \bar{x} \rightsquigarrow \bar{x}$.

5.6.2 Abstract Domain

In this section we describe the abstract domain for functional dependency analysis. Also the partial ordering over abstract states, and the least upper bound operator is explained.

The abstract store is a set of *lookups*, which are defined by the following *lookup function*.

Definition 35 (Lookup Function) *The* lookup function $\mathit{lookup}(p, n, K)$*, where p is a predicate symbol, n is the arity of p and $K \subseteq \{1, ..., n\}$ a set of integers, is*

defined as follows.

$$\begin{array}{lll} \mathsf{lookup}(p,n,K) & = p(\mathsf{lookup}(1,K),...,\mathsf{lookup}(n,K)) & \\ \mathsf{lookup}(i,K) & = * & i \in K \\ \mathsf{lookup}(i,K) & = \_ & i \notin K \end{array}$$

□

Consider the constraint p/2, then the set of possible lookups are

$$\{p(*,*), p(*,\_), p(\_,*), p(\_,\_)\}$$

Note that $\mathsf{lookup}(p,n,K)$ is isomorphic to K, and its usage is mainly syntactic. The concept of a *lookup* will become relevant to CHR optimisation. The set of arguments represented by the $*$s are referred to as the *key* of the lookup.

We can now define an abstraction function α_{fd-S} over the CHR store.

Definition 36 *Let S be a CHR store, and let $\mathtt{p}/n$ be the functor/arity of a CHR constraint of interest (e.g. any CHR constraint appearing in program P). Let $K \subseteq \{1,...,n\}$ be a set of integers, let $c = \mathsf{lookup}(p,n,K)$, let $count = \mathsf{count}(p,n,K,S,B)$ and let $count' = \mathsf{count}(p,n,K,S',B)$ where S' is defined as follows. Let $p(x_1,...x_n)\#i : j$ be the first active constraint in A with predicate p, then $S' = S - \{p(x_1,...x_n)\#i\}$, otherwise (if no such active constraint exists) $S' = S$. Then*

- $c^0 \in \alpha_{fd-S}(A,S,B)$ *if* $count = 0$
- *if* $count = 1$ *then*
  - $c^{1a} \in \alpha_{fd-S}(A,S,B)$ *if* $count' = 0$
  - $c^1 \in \alpha_{fd-S}(A,S,B)$ *otherwise*
- *if* $count = 2$ *then*
  - $c^{2a} \in \alpha_{fd-S}(A,S,B)$ *if* $count' = 1$
  - $c^* \in \alpha_{fd-S}(A,S,B)$ *otherwise*
- $c^* \in \alpha_{fd-S}(A,S,B)$ *otherwise.*

We define $\alpha_{fd-S}(A,S,B)$ to be the smallest possible set satisfying the above conditions. □

We refer the superscript associated with each lookup as the *counter* for that lookup. The counters 0, 1 and $*$ are fairly intuitive, as they mean that there is at most 0, 1 or many constraints in the CHR store with the same key as the lookup. The special counters, $1a$ and $2a$ are slightly more complicated. The counter $1a$ is equivalent to 1 except that if we were to remove the top-most active constraint of the same functor/arity from the concrete store, then the new counter will be

0. Similarly, the counter $2a$ is equivalent to 2 (although we treat 2 the same as $*$), but if the top-most active constraint were to be removed, then the counter will be 1. We shall refer to these special counters as *marked counters*, and other counters as *unmarked counters*. Marked counters are necessary since functional dependency analysis relies on improving the counters (i.e. moving to a lower counter) if possible.

Example 52 *Consider the following CHR store from the* ***database*** *program from Example 15.*

$$S = \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}$$

Assume that the built-in store is trivial, i.e. $B = true$*, and the given execution stack is* $A = [\texttt{entry}(key, cat)\#2 : 1]$*, then*

$$S' = \{\texttt{entry}(key, dog)\#1\}$$

hence

$$\alpha_{fd-S}(A, S, B) = \{\texttt{entry}(*, *)^1, \texttt{entry}(*, \_)^{2a}, \texttt{entry}(\_, *)^1, \texttt{entry}(\_, \_)^{2a}\}$$

Both $\texttt{entry}(*, *)$ *and* $\texttt{entry}(\_, *)$ *have the counter* 1*, since there is at most one* ***entry*** *that shares the same key* $\{1, 2\}$ *or* $\{2\}$*. One the other hand, both* $\texttt{entry}(*, \_)$ *and* $\texttt{entry}(\_, \_)$ *have the marked counter* $2a$*. This is because there are two* ***entry*** *constraints that share the same key* $\{1\}$ *or* $\emptyset$*, however there would be only one such constraint if we were to remove* $\texttt{entry}(key, cat)$ *(the current active constraint) from consideration.*

If the given activation stack had been empty, i.e. $A = []$*, then* $S = S'$ *hence*

$$\alpha_{fd-S}(A, S, B) = \{\texttt{entry}(*, *)^1, \texttt{entry}(*, \_)^*, \texttt{entry}(\_, *)^1, \texttt{entry}(\_, \_)^*\}$$

There are still 2 *constraints sharing the same key* $\{1\}$ *or* $\emptyset$*. However there is no current active constraint to remove from consideration, thus the counter for* $\texttt{entry}(*, \_)$ *and* $\texttt{entry}(\_, \_)$ *is now* $*$*.* □

We can now define the main abstraction function for set semantic functional dependencies.

Definition 37 (Abstraction Function) *We define function* α_{fd} *as follows*

$$\alpha_{fd}(\langle c, A, S, B, \_\rangle_{\_}) = \langle \alpha_{fd-c}(c), \alpha_{fd-S}([c|A], S, B)\rangle$$

where α_{fd-c} *is defined as*

$$\begin{array}{lll}
\alpha_{fd-c}(\Box) & = \Box & \\
\alpha_{fd-c}(c) & = \textbf{\textit{builtin}} & (c \textit{ built-in}) \\
\alpha_{fd-c}(p(_, ..., _)) & = p & \\
\alpha_{fd-c}(p(_, ..., _)\#i) & = p\# & \\
\alpha_{fd-c}(p(_, ..., _)\#i : j) & = p{:}j & \\
\alpha_{fd-c}([c|G]) & = [\alpha_{fd-c}(c)|\alpha_{fd-c}(G)] &
\end{array}$$

□

Example 53 *Consider the following execution state σ for the* `database` *program in Example 15.*

$$\langle \texttt{entry}(key, cat)\#2 : 1, [], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\} true, \emptyset \rangle_3$$

The CHR and built-in stores are the same as in Example 52. Then

$$\alpha_{fd}(\sigma) = \langle \texttt{entry}: 1, \{\texttt{entry}(*, *)^1, \texttt{entry}(*, \_)^{2a}, \texttt{entry}(\_, *)^1, \texttt{entry}(\_, \_)^*\} \rangle$$

□

We also require a partial order on abstract states.

Definition 38 (Partial Ordering) *Let $s_0 = \langle G_0, S_0 \rangle$ and $s_1 = \langle G_1, S_1 \rangle$, then $s_0 \preceq_{fd} s_1$ iff $G_0 = G_1$ and $S_0 \preceq_{fd_S} S_1$. The partial order $\preceq_{fd_S}$ over abstract CHR stores is defined as follows. If for all $c^n \in S_0$, there exists a $c^m \in S_1$ (with the same c), and $c^n \preceq c^m$, then $S_0 \preceq_{fd_S} S_1$. Here we define*

$$c^0 \prec c^{1a} \prec c^1 \prec c^{2a} \prec c^*$$

Otherwise $\preceq_{fd_S}$ is undefined if S_0 and S_1 contain different lookups.[5] □

We can use the definition of the partial ordering to define the concretisation function of this abstract domain as $\gamma_{fd}(s) = \{\sigma \mid \alpha_{fd}(\sigma) \preceq_{fd} s\}$.

The least upper bound operation over abstract stores is defined as follows.

Definition 39 (Least Upper Bound) *Let $s_0 = \langle G_0, S_0 \rangle$ and $s_1 = \langle G_1, S_1 \rangle$, then $s_0 \sqcup_{fd} s_1 = S_0 \sqcup_{fd-S} S_1$ if $G_0 = G_1$, otherwise it is undefined. The operator $\sqcup_{fd-S}$ over abstract CHR stores is defined as follows. If for all $c^n \in S_0$, there exists a $c^m \in S_1$ (with the same c), then $\mathsf{max}_{\preceq_{fd-S}}(c^n, c^m) \in S_0 \sqcup_{fd-S} S_1$. The set $S_0 \sqcup_{fd-S} S_1$ must be the minimal set satisfying the above condition. Here, function $\mathsf{max}_{\preceq_{fd-S}}$ is a maximum function using the ordering $\preceq_{fd-S}$ given in Definition 38. Otherwise $\sqcup_{fd-S}$ is undefined if S_0 and S_1 contain different lookups.* □

5.6.3 Abstract Transitions

Like before, the abstract transitions decide whether the transition is applicable to the current state s_0, and define the resulting state s afterwards.

We can now define the abstract transitions for functional dependency analysis.

Definition 40 (Abstract Transitions for Functional Dependencies)
1. AbstractSolve

$$s_0 = \langle \texttt{builtin}, S \rangle$$

[5] In the abstract interpretation, S_0 and S_1 will always have the same set of lookups.

Let S_g be the maximal subset of S such that for all $p(\_,\ldots,\_)^{\text{-}} \in S_g$ we have that $\mathsf{is\_ground}(p)$ holds. Note that the underscore superscript, e.g. $c^{\text{-}}$, is allowed to match any counter for c. Let $S_{ng} = S - S_g$, then

$$\begin{array}{rcl} S_0 & = & S_g \uplus \mathsf{multi}(S_{ng}) \\ s_j & = & \langle \Box, S_j \rangle = \sqcup_{fd}\{s_j^i \mid \ \langle p_i\#, S_{j-1}\rangle \rightarrowtail^* s_j^i \wedge \mathsf{final}(s_j^i) \wedge 1 \leq i \leq n\}, j \geq 1 \end{array}$$

where p_i are predicates of all potentially nonground constraints. Let k be the smallest positive integer such that $s_k = s_{k-1}$. Then $s = \langle \Box, S_k \rangle$.

Adding a built-in constraint, e.g. an equation to the built-in store has the potential to increase the counts of lookups for nonground constraints arbitrarily. Therefore, we use function multi on the nonground lookups, which overwrites any count by $*$.

$$\mathsf{multi}(S) = \{c^* \mid c^{\text{-}} \in S\}$$

In effect, we are assuming the weakest possible information for these lookups.

2. AbstractActivate

$$s_0 = \langle p, S\rangle \rightarrowtail \langle p{:}1, \mathsf{increase}(p, S)\rangle = s$$

A new constraint is added to the store, hence we must *increase* the counts for p. Here we define

$$\mathsf{increase}(p,S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l} \text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{increase}(c) \\ \text{otherwise } c' = c \end{array} \right\}$$

where

$$\begin{array}{ll} \mathsf{increase}(c^0) & = c^{1a} \\ \mathsf{increase}(c^{1a}) & = c^{2a} \\ \mathsf{increase}(c^1) & = c^{2a} \\ \mathsf{increase}(c^{2a}) & = c^* \\ \mathsf{increase}(c^*) & = c^* \end{array}$$

Notice that the resulting counts are always marked (except for $*$).

3. AbstractReactivate

$$s_0 = \langle p\#, S\rangle \rightarrowtail \langle p{:}1, S\rangle = s$$

Unlike AbstractActivate, the new active constraint is already present in the store, hence there is no need to call function $\mathsf{increase}$ on the abstract store.

4. AbstractDrop

$$s_0 = \langle p{:}j, S\rangle \rightarrowtail \langle \Box, \mathsf{unmark}(p, S)\rangle = s$$

No such occurrence j for predicate p.

Because the active constraint for p no longer exists, we must *unmark* all of the counters for p.

$$\mathsf{unmark}(p,S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l} \text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{unmark}(c) \\ \text{otherwise } c' = c \end{array} \right\}$$

where

$$\begin{aligned}\mathsf{unmark}(c^0) &= c^0\\ \mathsf{unmark}(c^{1a}) &= c^1\\ \mathsf{unmark}(c^1) &= c^1\\ \mathsf{unmark}(c^{2a}) &= c^*\\ \mathsf{unmark}(c^*) &= c^*\end{aligned}$$

In effect, the (former) active constraint is now treated the same as any other constraint in the store.

5. AbstractSimplify

$$s_0 = \langle p{:}\, j, S\rangle$$

Let r be the rule which contains the j^{th} occurrence of predicate p. If we assume that the rule fired, then

$$\langle \alpha_{fd-c}(C), \mathsf{decrease}(p, S)\rangle \rightarrowtail^* s_1 = \langle \Box, S_1\rangle$$

Because the active constraint has been deleted, we must *decrease* all of the counters for p.

$$\mathsf{decrease}(p, S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l}\text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{decrease}(c)\\ \text{otherwise } c' = c\end{array} \right\}$$

where

$$\begin{aligned}\mathsf{decrease}(c^0) &= c^0\\ \mathsf{decrease}(c^{1a}) &= c^0\\ \mathsf{decrease}(c^1) &= c^1\\ \mathsf{decrease}(c^{2a}) &= c^1\\ \mathsf{decrease}(c^*) &= c^*\end{aligned}$$

Note that we can only alter the *marked* counters. Also, the resulting counts are unmarked, since the active constraint has been deleted.

We consider the following three cases for deriving the resulting state s.

1. If rule r is of the form

$$p(x_1, ..., x_n)[\backslash,]p(y_1, ..., y_n)_j \Longleftrightarrow x_{i_0} = y_{i_0} \wedge ... \wedge x_{i_m} = y_{i_m} \mid C$$

(where occurrence j is shown).

If we assume the rule did not fire, then the resulting state is

$$s_2 = \langle p{:}\,(j+1), \mathsf{not\_fired}(p, \{i_0, ..., i_m\}, S)\rangle$$

We use function not_fired to improve the counts of lookups which contain key $\{i_0, ..., i_m\}$, where

$$\mathsf{not\_fired}(p, n, K, S) = \left\{ c' \;\middle|\; \begin{array}{l}c \in S \wedge\\ \text{if } \exists K' \supseteq K \text{ such that } c = \mathsf{lookup}(p, n, K')\\ \quad \text{then } c' = \mathsf{not\_fired}(c)\\ \text{otherwise } c' = c\end{array} \right\}$$

and

$$\begin{aligned}\mathsf{not\_fired}(c^0) &= c^0\\ \mathsf{not\_fired}(c^{1a}) &= c^{1a}\\ \mathsf{not\_fired}(c^1) &= c^1\\ \mathsf{not\_fired}(c^{2a}) &= c^1\\ \mathsf{not\_fired}(c^*) &= c^*\end{aligned}$$

We can make this improvement since if c^{2a} were in the abstract store for some lookup c with a key containing $\{i_0, ..., i_m\}$, then the rule *must* have fired. This kind of improvement is the essential part of functional dependency analysis.

For the resulting state we have $s = \mathtt{two}(s_1, s_2)$.

2. If rule r is an unconditional simplification rule (i.e. the guard is *true*) of the form

$$p(x_1, ..., x_n)_j \iff C$$

then $s = \mathtt{one}(s_1)$.

3. Otherwise (r is not in any of the above forms), then

$$s = \mathtt{two}(s_1, \langle p\!:\!(j+1), S\rangle)$$

6. AbstractPropagate

$$s_0 = \langle p\!:\!j, S\rangle$$

Applicable if occurrence j is a propagation occurrence

$$r \ @\ H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$$

Let

$$\begin{aligned}&S_0 = S\\ &\langle \alpha_{fd-c}(C), S_{j-1}\rangle \rightarrowtail^* \langle \Box, S_j\rangle = s_j\end{aligned}$$

Let k be the smallest positive integer such that $s_k = s_{k-1}$, then

$$s = \langle p\!:\!(j+1), S_k\rangle$$

7. AbstractGoal

$$s_0 = \langle [c|G], S\rangle \rightarrowtail \langle G, S'\rangle = s$$

where

$$\langle c, S\rangle \rightarrowtail^* \langle \Box, S'\rangle$$

□

For any given program point, we can determine which functional dependencies exist by interpreting the counts on the lookups. In general, a lookup with a count of 1 represents a functional dependency. For example, the lookup $\mathtt{p}(*, \_)^1$ indicates the functional dependency $p(x, y) :: \{x\} \rightsquigarrow \{x, y\}$ holds for the given program

point. The marked count of $2a$ can also represent a functional dependency if we also consider the results of late storage analysis. If, at a given program point, the active constraint has not been stored, then a lookup with a count of $2a$ can be treated the same as a count of 1. Otherwise, a count of $2a$ represents no functional dependency.

We may also derive bonus information from functional dependency analysis. It is possible that all lookups have a count of 0 at a given program point.[6] This means that no constraint associated with these lookups can be present in the CHR store for any corresponding concrete state. We call this property *never-stored*, and some optimisations are based on it. The interpretation of $1a$ is analogous to the interpretation of $2a$ if we take late storage into consideration: if the active constraint has not been stored, then $1a$ can be treated the same as 0.

5.6.4 Example Analysis

Consider the following additional rule on the `entry` constraint we could add to the `database` program in Example 15.

```
entry(X,Y)2 \ entry(X,Z)1 <=> Y=Z.
```

The abstract derivations for executing a single `entry` constraint are shown in Figure 5.3. For brevity, we have abbreviated `entry` to `e`, and we have omitted the subcomputations for the AbstractPropagate. In each instance, the AbstractPropagate does not change the abstract store. We are also assuming that the `entry` constraint is a ground constraint.

After three iterations we arrive at a fixed point for the final state. The resulting abstract store is

$$\{\texttt{entry}(*,*)^1, \texttt{entry}(*,\_)^1, \texttt{entry}(\_,*)^*, \texttt{entry}(\_,*)^*\}$$

This indicates that after an `entry` constraint has finished being active, the set semantic functional dependencies $\texttt{entry}(X,Y) :: \{X\} \rightsquigarrow \{X,Y\}$ and $\texttt{entry}(X,Y) :: \{X,Y\} \rightsquigarrow \{X,Y\}$ hold.

5.7 Summary

In this chapter we have presented a general abstract interpretation framework for the call-based operational semantics for CHRs. We have also presented two instances of the framework, which determine useful information from CHR programs that will be used in later chapters of this thesis.

In order to make abstract interpretation of CHRs feasible, it was necessary to construct a variant of the refined operational semantics. The call-based operational semantics was introduced so the number of abstract goals can be made

[6] Because of dependencies between lookups, if one lookup has a count of 0, then all lookups must have a count of 0.

$$
\begin{array}{ll}
 & \langle \mathsf{e}, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,\_)^0, \mathsf{e}(\_,*)^0, \mathsf{e}(\_,\_)^0\}\rangle \\
\rightarrowtail_{AA} & \langle \mathsf{e}:1, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle \\
 & \text{---------- first} \\
 & \rightarrowtail_{ASi} \quad \langle \square, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,\_)^0, \mathsf{e}(\_,*)^0, \mathsf{e}(\_,\_)^0\}\rangle \\
 & \text{---------- second} \\
 & \rightarrowtail_{\neg ASi} \quad \langle \mathsf{e}:2, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle \\
 & \rightarrowtail_{AP} \quad \langle \mathsf{e}:3, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,\_)^{1a}, \mathsf{e}(\_,*)^{1a}\mathsf{e}(\_,\_)^{1a}\}\rangle \\
 & \rightarrowtail_{ADp} \quad \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1, \mathsf{e}(\_,\_)^1\}\rangle \\
 & \text{---------- lub} \\
\rightarrowtail_{\sqcup} & \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1, \mathsf{e}(\_,\_)^1\}\rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathsf{e}, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1, \mathsf{e}(\_,\_)^1\}\rangle \\
\rightarrowtail_{AA} & \langle \mathsf{e}:1, \{\mathsf{e}(*,*)^{2a}, \mathsf{e}(*,\_)^{2a}, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle \\
 & \text{---------- first} \\
 & \rightarrowtail_{ASi} \quad \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^1, \mathsf{e}(\_,\_)^1\}\rangle \\
 & \text{---------- second} \\
 & \rightarrowtail_{\neg ASi} \quad \langle \mathsf{e}:2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle \\
 & \rightarrowtail_{AP} \quad \langle \mathsf{e}:3, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^{2a}\mathsf{e}(\_,\_)^{2a}\}\rangle \\
 & \rightarrowtail_{ADp} \quad \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle \\
 & \text{---------- lub} \\
\rightarrowtail_{\sqcup} & \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \mathsf{e}, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle \\
\rightarrowtail_{AA} & \langle \mathsf{e}:1, \{\mathsf{e}(*,*)^{2a}, \mathsf{e}(*,\_)^{2a}, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle \\
 & \text{---------- first} \\
 & \rightarrowtail_{ASi} \quad \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle \\
 & \text{---------- second} \\
 & \rightarrowtail_{\neg ASi} \quad \langle \mathsf{e}:2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle \\
 & \rightarrowtail_{AP} \quad \langle \mathsf{e}:3, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*\mathsf{e}(\_,\_)^*\}\rangle \\
 & \rightarrowtail_{ADp} \quad \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle \\
 & \text{---------- lub} \\
\rightarrowtail_{\sqcup} & \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,\_)^1, \mathsf{e}(\_,*)^*, \mathsf{e}(\_,\_)^*\}\rangle
\end{array}
$$

Figure 5.3: Example abstract derivation for functional dependency analysis

finite. Also, transitions under the call-based semantics moved from program point to program point, which further simplifies the framework. We presented a proof that the call-based semantics is equivalent to the original refined semantics.

Next we presented the abstract framework itself. The framework itself is fairly simple: starting from an initial abstract goal we continually apply abstract tran-

sitions until a final state is reached. The only exception to this is AbstractSimplify, which may return two states: $\texttt{two}(s_1, s_2)$, where s_1 is the state when the rule fired, and s_2 when the rule did not fire. The framework handles the split by calculating the final state for s_2, and calculating the lub with s_1.

We presented two non-trivial and useful instances of the framework. The first is *late storage analysis*, which determines when an active constraint needs to be stored based on when it is first *observed*. The second is *functional dependency analysis*, which attempts to discover functional dependency relationships between arguments of constraints in the CHR store. The results from both of these analyses will be used later in the thesis.

Not all analysis for CHRs has been formalised as instances of the framework, and some of these analyses will be mentioned later on. These include the analysis for continuation optimisation (see Chapter 7) and delay avoidance (see Chapter 8), etc. However, these types of analysis are less complex, hence a more ad hoc analysis is sufficient. These will be introduced as required.

Chapter 6

Confluence

6.1 Introduction

The refined operational semantics is still nondeterministic, hence the property of confluence is important. There are two sources of nondeterminism under the refined operational semantics. The first arises from the **Solve** transition, where the order in which the woken up constraints are (re)added to the execution stack is left open.

Example 54 *Consider the following state for the* `leq` *program.*

$$\langle [C = B], \{\texttt{leq}(C, A)\#2, \texttt{leq}(A, B)\#1\}, true\rangle_5$$

Assume that the wakeup policy includes all non-fixed CHR constraints. ***Solve*** *requires constraints* $\texttt{leq}(C, A)$ *and* $\texttt{leq}(A, B)$ *to be (re)added to execution stack. The order is arbitrary, hence*

$$\langle [\texttt{leq}(C, A)\#2, \texttt{leq}(A, C)\#1], \{\texttt{leq}(C, A)\#2, \texttt{leq}(A, B)\#1\}, C = B\rangle_5$$

or

$$\langle [\texttt{leq}(A, C)\#1, \texttt{leq}(C, A)\#2], \{\texttt{leq}(C, A)\#2, \texttt{leq}(A, B)\#1\}, C = B\rangle_5$$

are equally valid states after applying ***Solve****.* □

In this example the choice of the states after **Solve** is inconsequential, because the `leq` program is confluent under the refined semantics.

The other source of nondeterminism arises the **Simplify** and **Propagate** transitions, which do not specify which partner constraints (i.e. H_1, H_2 and H_3 from Definition 11) should be chosen for the transition (if more than one possibility exists). Example 25 shows how this choice can result in two non-variant final states.

Both sources of nondeterminism could be removed by further "refining" the operational semantics. For example, we could impose an order on matchings for

Simplify and **Propagate**, or an order on the constraints woken up after **Solve**. The advantage is that all programs are trivially confluent under a deterministic operational semantics.

There are two main reasons against this idea. The first is that different CHR implementations use different data structures to (efficiently) represent the store, and this usually affects the order partner constraints are matched against the head of a rule. By imposing an artificial order on partner constraints may have an adverse effect on efficiency, since we have restricted or complicated the data structures that can be used. The second reason is that it not clear how further refining the semantics benefits the programmer. Many CHR programs have already been implemented using the refined operational semantics without any additional assumptions about orderings, etc. Therefore if a program is not confluent under the refined semantics then this generally indicates a bug, and ideally the compiler should detect this if possible.

We provided a theoretical result in Chapter 3 which ensures that terminating (under ω_r) and confluent (under ω_t) programs are confluent under the refined semantics. Sometimes this is useful, e.g. with the `leq` program, but in general programs under the refined semantics are not confluent under the theoretical semantics. This directly follows from the fact that the refined semantics is more deterministic. In this chapter we look at testing for confluence under the refined semantics alone. We propose several static analyses designed to detect non-confluent programs. A confluence checker has been implemented in the HAL CHR compiler, and we test it on several "large" CHR programs.

The rest of the chapter is structured as follows. Section 6.2 presents the theory behind the confluence check. Section 6.3 shows how the confluence test can be implemented inside a CHR compiler. Section 6.4 uses the confluence test on three large CHR programs. Finally, we conclude.

6.2 Checking Confluence for ω_r

Confluence is tested under the theoretical operational semantics by calculating all critical pairs between rules, and showing that these critical pairs are joinable (see Section 2.3.2 or [32, 1] for details). It is tempting to think that this could be adapted to the refined operational semantics, e.g. by checking joinability of all critical pairs between rules and themselves,[1] however this is not the case.

Example 55 *Consider the following (somewhat contrived) CHR program.*

```
r1 @ p, q(_) <=> r.
r2 @ q(_), q(_) \ r <=> true.
r3 @ r \ q(_) <=> true.
r4 @ r <=> true.
```

[1] Under the refined operational semantics, critical pairs between rules makes no sense, since it is impossible to construct a state that can fire two distinct rules.

The following (reachable) state in analogous to the non-trivial direct common ancestor state of `r1` *and itself*

$$\sigma = \langle [p\#3 : 1], \{p\#3, q(A)\#2, q(B)\#1\}\rangle_4$$

This state is reachable from the initial goal $[q(B), q(A), p]$*. No matter the derivation chosen, this state will always be reduced to the empty state* $\langle [], \emptyset\rangle_5$*, hence it appears that the critical pair for* `r1` *and itself is joinable. We can similarly verify all other rules in this program.*

Unfortunately, the program is still not confluent, as is easily demonstrated by the following (reachable) state.

$$\sigma' = \langle [p\#4 : 1], \{p\#4, q(A)\#3, q(B)\#2, q(C)\#1\}\rangle_5$$

This state is reachable from the initial goal $[q(C), q(B), q(A), p]$*. The two possible (distinct) final states are* $\langle [], \{q(A)\#3, q(B)\#2\}\rangle_6$ *and* $\langle [], \{q(B)\#2, q(C)\#1\}\rangle_6$ *because of three possible matches for* $q(A)$ *in the first state. We note that state* σ' *is not a non-trivial direct common ancestor state of any rule in the program.* □

The example illustrates how extending the notion of critical pairs does not appear to work under the refined operational semantics. Under the theoretical semantics more critical pairs would be considered, i.e. critical pairs between different rules, and non-confluence will be detected (e.g. consider the critical pair between rule `r1` and `r3`, etc.).

We present a different approach, based on the sources of nondeterminism under the refined semantics. We will identify some conditions, which if satisfied, guarantee confluence.

6.2.1 Nondeterminism in the Solve Transition

The first source of nondeterminism under the refined semantics occurs when deciding the order on the the set of woken up constraints during a **Solve** transition. To avoid this nondeterminism we will require this set to be empty.[2] This is a very common case, as it occurs when the all CHR constraints are fixed/ground at runtime. We will generalise this slightly by imposing conditions on the wakeup policy used by the implementation.

To formalise this we define the *trivial wakeup policy* that does not wakeup any CHR constraint on a **Solve** transition.

Definition 41 (Trivial Wakeup Policy) *Given a CHR store* S*, built-in constraint* c *and built-in store* B*, we define the* trivial wakeup policy *as*

$$trivial(S, c, B) = \emptyset$$

□

[2] Another possibility is to require this set to be singleton.

For $trivial(S, c, B)$ to satisfy the definition of a wakeup policy (see Definition 10), the constraints in S must always be fixed.[3] The HAL CHR compiler determines this information from `mode` declarations, i.e. a constraint will always be fixed if each argument has the declared mode of '`in`'.

For programs that really do interact with a built-in constraint solver (e.g. the `leq` solver from Example 1), we currently have no better test other than relying on the confluence test of the theoretical operational semantics. In this case it is very hard to see how the programmer can control execution sufficiently.

6.2.2 Nondeterminism in the Simplify and Propagate Transitions

The second source of nondeterminism occurs when there is more than one set of partner constraints in the CHR store that can be matched against when applying the **Simplify** or **Propagate** transitions.

We formalise this as follows. A *matching* is a sequence of numbered constraints from the CHR store that match with the head of a rule when applying **Simplify** or **Propagate**.

Definition 42 (Matching) *A* matching M *of occurrence* j *with active CHR constraint* c *in state* $\langle [c\#i : j|A], S, B, T\rangle_n$ *is a named tuple of numbered constraints from* S *that match against the head of rule* r *of occurrence* j*. These are*

$$M = prop(H_1, c\#i, H_2, H_3) \qquad \textit{for } \textbf{Propagate}$$
$$M = simp(H_1, H_2, c\#i, H_3) \qquad \textit{for } \textbf{Simplify}$$

where H_1*,* H_2 *and* H_3 *are the matching constraints as defined by Definition 11 (the refined semantics).* □

Note that the order of the constraints in a matching M exactly corresponds with the order of the constraints in the rule that matched with it.

Most of the time we will treat matchings as sequences or multisets. For example, the matching $prop(H_1, c\#i, H_2, H_3)$ can be treated as the sequence $(H_1 +\!\!+ [c\#i] +\!\!+ H_2 +\!\!+ H_3)$. Similarly, $simp(H_1, H_2, c\#i, H_3)$ is treated as $(H_1 +\!\!+ H_2 +\!\!+ [c\#i] +\!\!+ H_3)$.

The definition of the refined operational semantics does not specify which matching to choose if more than one is available. Non-confluence arises when given a state σ, there are more than one possible matchings M_1 and M_2 (w.r.t. some rule r) such that firing r on M_1 results in a different answer than firing r on M_2.

To help simplify things further, we define the following helper functions which map matchings to useful information about matchings. These will be useful later.

[3] There may be other restrictive circumstances where the usage of $trivial(S, c, B)$ as a wakeup policy is correct.

We define function $delete(M)$ which returns the multiset of constraints in M which are deleted by occurrence, i.e.

$$delete(prop(H_1, c\#i, H_2, H_3)) = H_3$$
$$delete(simp(H_1, H_2, c\#i, H_3)) = H_2 \uplus \{c\#i\} \uplus H_3$$

We also define $entry(r, M)$ which returns the propagation history entry associated with a rule r and a matching M, i.e.

$$entry(r, prop(H_1, c\#i, H_2, H_3)) = ids(H_1) +\!\!+ [i] +\!\!+ ids(H_2) +\!\!+ ids(H_3) +\!\!+ [r]$$
$$entry(r, simp(H_1, H_2, c\#i, H_3)) = ids(H_1) +\!\!+ ids(H_2) +\!\!+ [i] +\!\!+ ids(H_3) +\!\!+ [r]$$

We also similarly define $\theta(r, M)$ to be the matching substitution, such that M is unified with the head of r, and

$$goal(r, M) = \theta(r, M)(C) \qquad \text{where } r = (H_1 \backslash H_2 \iff g \mid C)$$

to be the body of the renamed rule used by the transition. Using these functions, we can write the result of applying **Simplify** or **Propagate** to a state $\langle [c\#i : j|A], S, B, T\rangle_n$ and matching $M \subseteq S$ as as $\langle goal(r, M) +\!\!+ A, S - delete(M), B, \{entry(r, M)\} \cup T\rangle_n$ where r is the renamed copy of the rule used by the transition.

Non-confluence can arise when multiple matchings exist for a rule r, and r is not allowed to eventually try them all. This may happen when firing r with one matching results in the deletion of a constraint in another matching.

Definition 43 (Matching Completeness) *An occurrence j in (renamed) rule r is* matching complete *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, ..., M_m$ possible matchings, then for all $M_i \in \{M_1, ..., M_m\}$ if*

$$\langle goal(r, M_i), S - delete(M_i), B, \{entry(r, M_i)\} \cup T\rangle_n \rightarrowtail^* \langle A', S', B', T'\rangle_{n'}$$

then for all $M_j \in \{M_1, ..., M_m\} - \{M_i\}$ we have that $M_j \subseteq S'$. □

In other words, firing rule r for any matching M_i and executing $goal(r, M_i)$ does not result in the deletion of a constraint occurring in a different matching $M_k, k \neq i$. The intention is that rules will always try all possible matchings unless failure occurs.

Note that r itself may directly delete the active constraint (via the **Simplify** transition). If so, r will only be matching complete if there is only one possible matching, i.e., $m = 1$.

Example 56 *Consider the* `database` *confluence problem from Example 25. This can be expressed as a matching completeness problem since there exists a state, namely*

$$\langle [\texttt{lookup}(key, V)\#3 : 1], \{\texttt{lookup}(key, V)\#3, \texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, true\rangle_4$$

with two matchings $M_1 = [\texttt{entry}(key, cat)\#2, \texttt{lookup}(key, V)\#3]$ *and* $M_2 = [\texttt{entry}(key, dog)\#1, \texttt{lookup}(key, V)\#3]$ *such that firing the rule on* M_1 *deletes constraint* $\texttt{lookup}(key, V)\#3$ *(the active constraint) which also appears in* M_2*. Therefore the occurrence for* ***lookup*** *cannot be matching complete.*

This occurrence will be matching complete if all states where there are multiple matchings for a given lookup are unreachable. This can be achieved by adding a rule that enforces a functional dependency, e.g. by adding the following rule to the start of the program.

```
killdup @ entry(Key,Val1) \ entry(Key,Val2) <=> Val1 = Val2.
```

This rule throws away duplicate entries for the same key. Now the occurrence is matching complete, since only one matching will ever be possible. □

Matching completeness can also be broken if the body of a rule indirectly deletes constraints from other matchings.

Example 57 *Consider the following CHR program*

```
r1 @ p1, q(X) ==> r(X).
r2 @ p2, r(a) <=> true.
```

The occurrence 1 *of* ***p*** *in* ***r1*** *is not matching complete because of the (reachable) state*

$$\langle [\texttt{p}\#3 : 1], \{\texttt{p}\#3, \texttt{q(a)}\#2, \texttt{q(b)}\#1\}\rangle_4$$

with matchings $M_1 = [\texttt{p}\#3, \texttt{q(a)}\#2]$ *and* $M_2 = [\texttt{p}\#3, \texttt{q(b)}\#1]$*. Firing* `r1` *against* M_1 *calls the new constraint* $\texttt{r}(a)$ *which in turn deletes* `p#3` *(which appears in both matchings) by firing rule* `r2`*. Therefore occurrence* 1 *for* `p` *is not matching complete.* □

A matching complete occurrence is guaranteed to eventually try all possible matchings for a given execution state. However, matching completeness is sometimes too strong if the programmer does not care which matching is chosen. This is common when the rule body does not depend on the matching.

Example 58 *For example, consider the following rule from a simple ray tracer (see Appendix A.2).*

```
shadow @ sphere(C,R,_) \ light_ray(L,P,_,_) <=>
                                    blocks(L,P,C,R) | true.
```

This rule calculates if point `P` *is in shadow by testing if the ray from light* `L` *is blocked by a sphere at* `C` *with radius* `R`*. Consider an active* `light_ray` *constraint: there may be more than one* `sphere` *blocking the ray, however we do not care* which *sphere blocks, just* if *there is a sphere which blocks. This rule is not matching complete, but since the matching chosen does not affect the resulting state, it is matching independent.* □

We define *matching independence* as the property that the matching chosen does not matter.

Definition 44 (Matching Independence) *A matching incomplete occurrence for (renamed) rule r that deletes the active constraint only is* matching independent *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, \ldots, M_m$ possible matchings, then all of*

$$\langle goal(r, M_i), S - delete(M_i), B, \{entry(r, M_i)\} \cup T\rangle_n$$

for each $M_i \in \{M_1, ..., M_m\}$ are joinable (see Definition 20). □

The rule `shadow` in Example 58 satisfies the definition since $goal(r, M) = true$ for all possible matchings M, i.e. the goal does not depend on the matching chosen.

Suppose that a rule is matching complete, and there are multiple possible matchings. The ordering in which the matchings are tried is still chosen nondeterministically. Hence, there is still potential of non-confluence. For this reason we also require *order independence*, which ensures the choice of order does not affect the result.

Definition 45 (Order Independence) *A matching complete occurrence j in rule r is* order independent *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, \ldots, M_m$ possible matchings, the states*

$$\langle goal(r_i, M_i), S_j - delete(M_i), B_j, \{entry(r_i, M_i)\} \cup T_j\rangle_{n_j}$$

and

$$\langle goal(r_j, M_j), S_i - delete(M_j), B_i, \{entry(r_j, M_j)\} \cup T_i\rangle_{n_i}$$

(where r_i and r_j are distinct renamings of r) are joinable for all $M_i, M_j \in \{M_1, \ldots, M_m\}$ where S_i, S_j, B_i, B_j, T_i, T_j, n_i and n_j are given by final states arising from subcomputations

$$\langle goal(r_i, M_i), S - delete(M_i), B, \{entry(r_i, M_i)\} \cup T\rangle_n \rightarrowtail^* \langle A_i, S_i, B_i, T_i\rangle_{n_i} = \sigma_i$$

and

$$\langle goal(r_j, M_j), S - delete(M_j), B, \{entry(r_j, M_j)\} \cup T\rangle_n \rightarrowtail^* \langle A_j, S_j, B_j, T_j\rangle_{n_j} = \sigma_j$$

where σ_i and σ_j are final states. □

The following is a typical example of order independence.

Example 59 *Consider the following fragment of code for summing colours from the ray tracer.*

```
add1 @ add_color(C1), color(C2) <=> C3 = C1 + C2, color(C3).
add2 @ add_color(C) <=> color(C).
```

Assume the colours are encoded as ordinary integers (e.g. for a gray scale image). All occurrences of `color` *and* `add_color` *are matching complete. Furthermore, calling* `add_color`(C_1), ..., `add_color`(C_n) *results in* `color`(C_1 + ... + C_n). *Since addition is symmetric and associative, it does not matter in what order the* `add_color` *constraints are called. Consider the occurrence of* `output` *in*

```
render @ output(P) \ light_ray(_,P,C,_) <=> add_color(C).
```

Here, calling `output`(P) *calculates the (accumulated) color at point P where any* `light_ray`*s (a ray from a light source) may intersect. If there are multiple light sources, then there may be multiple* `light_ray` *constraints. The order* `add_color` *is called does not matter, hence the occurrence is order independent.* □

6.2.3 Confluence Test

We claim is that if a program P can is shown to satisfy the conditions outlined above, then it is confluent. In this section we present a formal proof of this fact.

Before we present the main result, we prove two useful lemmas.

Lemma 11 (Parallel Derivations I) *For all execution stacks A_1 and A_2 the following holds: $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle G' +\!\!+ A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ iff $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle G' +\!\!+ A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ or both states σ_k and σ'_k are $false$, provided no states in either derivation is of the form $\langle A_1, S', B', T'\rangle_{n'}$ or $\langle A_2, S', B', T'\rangle_{n'}$ respectively.*

Proof. Note that it suffices to prove one direction of the "iff" only, since the other direction is symmetric (i.e. obtained by substituting A_1 with A_2 and vice-versa). We prove the " $\Longrightarrow$ " direction by induction over derivations of length k.

Base case: Derivations of zero length ($k = 0$). Then $\sigma_0 = \sigma$ and $\sigma'_0 = \sigma$ are zero length derivation of the required form.

Induction step: Assume that for all derivations of length k that if $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle G_k +\!\!+ A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ then for all A_2 we have that $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle G_k +\!\!+ A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$. We show the same holds for derivations of length $k + 1$.

We consider all ω_r derivation steps from σ_k to σ_{k+1} and show the same derivation step can be applied to σ'_k to derive σ'_{k+1} of the required form.

By assumption G_k is non-empty, otherwise σ_k is of the form $\langle A_1, S', B', T'\rangle_{n'}$ which is not allowed. Therefore the top-most constraint on the respective execution stacks for σ_k and σ'_k are the same.

The $k + 1$ case easily verified by inspection over all of the transition steps for the refined operational semantics (Definition 11). All of these transitions only depend on the top-most constraint of the execution stack, and all transition preserve the tail of the execution stack. Thus, if $\sigma_k \rightarrowtail \sigma_{k+1}$ then $\sigma'_k \rightarrowtail \sigma'_{k+1}$ by the same transition step.

Therefore if $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle G_k +\!\!+ A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ then for all A_2 we have that $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle G_k +\!\!+ A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ provided the conditions noted in the Lemma above hold. By symmetry the other direction of the "iff" also holds. □

This next Lemma is almost identical to the previous one, except that it handles the case where all of goal G has finished executing.

Lemma 12 (Parallel Derivations II) *For all execution stacks A_1 and A_2 the following holds: $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ iff $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ or both states σ_k and σ'_k are $false$, provided no states in either derivation (apart from σ_k and σ'_k) are of the form $\langle A_1, S', B', T'\rangle_{n'}$ or $\langle A_2, S', B', T'\rangle_{n'}$ respectively.*

Proof. As with the proof of Lemma 12, it suffices to prove one direction of the "iff" only, since the other direction is symmetric. We prove the "$\Longrightarrow$" direction by direct proof.

There are two cases to consider. The first is that the derivation is of zero length, i.e. $\sigma = \sigma_k$, then $\sigma'_k = \sigma'$ satisfies the hypothesis.

The second case is derivations of non-zero length. Let D_k be the derivation $\sigma \rightarrowtail \sigma_k$ above. We can write $D_k = D_{k-1} \rightarrowtail \sigma_k$, where the last state in D_k is $\sigma_{k-1} = \langle G +\!\!+ A_1, S_{k-1}, B_{k-1}, T_{k-1}\rangle_{n_{k-1}}$ for some non-empty G. By Lemma 11 there is a derivation from σ' to the state $\sigma'_{k-1} = \langle G +\!\!+ A_2, S_{k-1}, B_{k-1}, T_{k-1}\rangle_{n_{k-1}}$. Call this derivation D_{k-1}.

Consider the transition from σ_{k-1} to σ_k. We can apply exactly the same transition to σ'_{k-1} to derive σ'_k of the above form (using the same argument as in the proof of Lemma 12).

Therefore if $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ then for all A_2 we have that $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ provided the conditions noted in the Lemma above hold. By symmetry the other direction of the "iff" also holds. □

We are ready for the main result. First we give a formal definition of the confluence test.

Definition 46 (Confluence Test) *A program P passes our* confluence test *if*

1. *P is terminating;*
2. *All occurrences in P are matching complete or matching independent; and*
3. *All matching complete occurrences in P are order independent.*

Also, the implementation uses $trivial(S, c, B)$ as the wakeup policy. □

We show that the test outlined above actually proves confluence. First we show that it at least proves *local confluence*, which is a weaker form of confluence.

Definition 47 (Local Confluence) *A CHR program is* local confluent *if the following holds for all states σ_0, σ_1 and σ_2 where σ_0 is a reachable state: If $\sigma_0 \rightarrowtail \sigma_1$ and $\sigma_0 \rightarrowtail \sigma_2$ then σ_1 and σ_2 are joinable with respect to σ_0.* □

The only difference between local confluence and confluence is that states σ_1 and σ_2 are derived after a single transition step, rather than an arbitrary number of steps. We can now state the Lemma.

Lemma 13 (Local Confluence Test) *Let P be a CHR program that satisfies Definition 46, then P is locally confluent.*

Proof. Direct proof. We show that all reachable states σ such that if $\sigma \rightarrowtail_1 \sigma_1$ and $\sigma \rightarrowtail_2 \sigma_2$ then σ_1 and σ_2 are joinable. Note the notation $\rightarrowtail_1$ and $\rightarrowtail_2$ representing the transitions from σ to σ_1 and σ_2 respectively.

By inspection, all of the conditions for ω_r transitions are pairwise mutually exclusive. In other words, it is not possible that $\rightarrowtail_1$ and $\rightarrowtail_2$ are different transitions, thus $\rightarrowtail_2 = \rightarrowtail_1$.

Assume $\rightarrowtail_1$ and $\rightarrowtail_2$ are one of **Activate**, **Reactivate**, **Drop** or **Default**. By inspection, all of these transitions are deterministic, hence $\sigma_1 = \sigma_2$ thus the two states are trivially joinable.

The remaining cases for $\rightarrowtail_1$ (and $\rightarrowtail_2$) are as follows.

CASE **Solve**:

Then σ is of the form $\langle [c|A], S, B, T\rangle_n$, where c is a built-in constraint, and σ_1 and σ_2 are both of the form

$$\langle trivial(S, c, B) +\!\!+ A, S, c \wedge B, T\rangle_n = \langle A, S, c \wedge B, T\rangle_n$$

The other case is that $\sigma_1 = \sigma_2 = false$. Either way $\sigma_1 = \sigma_2$ and hence are trivially joinable.

CASE **Simplify**:

State σ is of the form $\langle [c\#i : j|A], S, B, T\rangle_n$ and there are two (possibly identical) matchings M_1 and M_2 which satisfy the conditions for **Simplify**. Then σ_1 and σ_2 are given by

$$\sigma_m = \langle goal(r_m, M_m) +\!\!+ A, S - delete(M_m), B, \{entry(r_m, M_m)\} \cup T\rangle_n$$

for $m = 1$ and $m = 2$ respectively. Here, r_1 and r_2 are two distinct renamings of the rule used in the transition.

There are two possible cases to consider for the occurrence j.

1. j is *matching complete*: By the definition of **Simplify** the active constraint $c\#i$ is deleted, thus $c\#i \in delete(M_1)$ and $c\#i \in delete(M_2)$. Thus the only way for occurrence j to be matching complete is that there is only one possible matching, i.e. $M_1 = M_2$. Then σ_1 and σ_2 must be variants and therefore are trivially joinable.

2. j is *matching independent* (and matching incomplete): Matching independence requires the states given by

$$\sigma'_m = \langle goal(r_m, M_m), S - delete(M_m), B, \\ \{entry(r_m, M_m)\} \cup T \rangle_n$$

for $m = 1$ and $m = 2$ are joinable. This means that there exists variant states σ''_1 and σ''_2 such that $\sigma'_1 \rightarrowtail^* \sigma''_1$ and $\sigma'_2 \rightarrowtail^* \sigma''_2$. We write σ''_1 and σ''_2 as

$$\sigma''_m = \langle A''_m, S''_m, B''_m, T''_m \rangle_{n''_m}$$

for $m = 1$ and $m = 2$. Then by Lemma 12, we have that $\sigma_1 \rightarrowtail^* \sigma_3$ and $\sigma_2 \rightarrowtail^* \sigma_4$ where σ_3 and σ_4 are given by

$$\sigma_m = \langle A''_m +\!\!+ A, S''_m, B''_m, T''_m \rangle_{n''_m}$$

for $m = 3$ and $m = 4$. Clearly if σ''_1 and σ''_2 are variants then σ_3 and σ_4 are variants, therefore σ_1 and σ_2 are joinable.

CASE **Propagate**:

Matching independence is not applicable because **Propagate** does not delete the active constraint.

State σ is of the form $\langle [c\#i : j|A], S, B, T \rangle_n$ and there are two (possibly identical) matchings M_1 and M_2 which satisfy the conditions for **Propagate**. Then σ_1 and σ_2 are given by

$$\sigma_m = \langle goal(r_m, M_m) +\!\!+ [c\#i : j|A], S - delete(M_m), \\ B, \{entry(r_m, M_m)\} \cup T \rangle_n$$

for $m = 1$ and $m = 2$ respectively. Once again, r_1 and r_2 are two distinct renamings of the rule used in the transition.

Thanks to order independence, we know that the states given by

$$\sigma_{(m_1,m_2)} = \langle goal(r_{m_1}, M_{m_1}), S_{m_2} - delete(M_{m_1}), \\ B_{m_2}, \{entry(r_{m_1}, M_{m_1})\} \cup T_{m_2} \rangle_{n_{m_2}}$$

are joinable where S_{m_2}, B_{m_2}, T_{m_2} and n_{m_2} are given by final states arising from

$$\langle goal(r_{m_2}, M_{m_2}), S - delete(M_{m_2}), B, \{entry(r_{m_2}, M_{m_2})\} \cup T \rangle_n \rightarrowtail^* \\ \langle A_{m_2}, S_{m_2}, B_{m_2}, T_{m_2} \rangle_{n_{m_2}}$$

for $(m_1, m_2) = (1, 2)$ and $(m_1, m_2) = (2, 1)$. This means that there exists variant states σ''_1 and σ''_2 such that $\sigma_{(1,2)} \rightarrowtail^* \sigma''_1$ and $\sigma_{(2,1)} \rightarrowtail^* \sigma''_2$. We write σ''_1 and σ''_2 as

$$\sigma''_m = \langle A''_m, S''_m, B''_m, T''_m \rangle_{n''_m}$$

for $m = 1$ and $m = 2$.

Consider σ_1 defined above. Then by Lemma 1 we have that

$$\sigma_1 \rightarrowtail^* \langle [c\#i : j|A], S_1, B_1, T_1 \rangle_{n_1} = \sigma_1'$$

By matching completeness $M_2 \subseteq S_1$. W.l.o.g. we can assume $entry(r_2, M_2) \notin T_1$ and the guard still holds (because the built-in store in monotonic by assumption). Thus

$$\sigma_1' \rightarrowtail_{propagate} \langle goal(r_2, M_2) +\!\!+ [c\#i : j|A], S_1 - delete(M_2), \\ B_1, \{entry(r_2, M_2)\} \cup T_1 \rangle_{n_1}$$

By Lemma 12 we have that

$$\sigma_1 \rightarrowtail^* \sigma_1' \rightarrowtail^* = \langle A_1'' +\!\!+ [c\#i : j|A], S_1'', B_1'', T_1'' \rangle_{n_1''}$$

We can apply a symmetric argument to similarly derive

$$\sigma_2 \rightarrowtail^* \langle A_2'' +\!\!+ [c\#i : j|A], S_2'', B_2'', T_2'' \rangle_{n_2''}$$

These states must be variants because σ_1'' and σ_2'' (defined above) are also variants. Therefore σ_1 and σ_2 are joinable.

We have shown that if $\sigma \rightarrowtail \sigma_1$ and $\sigma \rightarrowtail \sigma_2$ then σ_1 and σ_2 are joinable. Therefore P is locally confluent. □

Finally, we can state the main result.

Theorem 5 (Confluence Test) *Let P be a CHR program that satisfies Definition 46, then P is confluent.*

Proof. By Lemma 13 program P is locally confluent. By definition P is terminating. Therefore by Newman's Lemma [64] P is confluent. □

6.3 Implementation of Confluence Test

So far we have introduced some conditions, e.g. matching completeness etc., and shown that if these conditions hold for a given program P, then P is confluent. The confluence test is undecidable in general, since it relies on termination, however in this section we discuss how a modern CHR compiler can test (with some assumptions) if these conditions hold based on information it collects from program analysis discussed in Chapter 5. We allow the tests to be inaccurate, in that it is allowed to reject programs that are confluent, but not the other way around. Later in this chapter we try the confluence tests on several examples in order to estimate accuracy.

The tests outlined below have been implemented as part of the HAL CHR compiler, which we will refer to as *confluence checker* from now on. The confluence checker implements partial tests for fixedness of CHR constraints, matching

completeness and matching independence, and relies on user annotation for determining order independence except for a few cases discussed below. The confluence checker assumes termination, which (as usual) is left to the programmer to decide.

The first part of the confluence test the HAL CHR compiler tests for is groundness/fixedness, since this is required for the usage of the trivial wakeup policy to be correct. The HAL compiler already has access to this information, since the user must write a `mode` declaration for each CHR constraint. If the modes for every argument for each CHR constraint are '`in`', then the program passes this part of the confluence test.

In HAL CHR, constraints are allowed to have mode '`out`' under restricted conditions. Let c be a CHR constraint with an '`out`' argument represented by v (which must be a '`new`' variable at runtime). The restrictions are:

1. c is *never-stored* anywhere in the program (see Section 5.6.3);
2. all possible rule bodies called by c (by firing a rule) either bind v to a ground value or fail.

The never-stored requirement will ensures that c is never in the CHR store whenever a built-in constraint is **Solve**d, hence we avoid the nondeterminism.

Example 60 *The `lookup(Key,Val)` constraint in Example 15 is a classic example of a constraint with a argument with mode '`out`'. Its mode declaration provided by the programmer is as follows.*

```
:- mode lookup(in,out) is semidet.
```

When called, variable `Val` will be '`new`', but will be bound to the corresponding value for the given key if it exists, or otherwise failure occurs. □

Since constraints with '`out`' modes can never be woken up, the usage of the trivial wakeup policy is still correct. Therefore, such programs also pass the confluence checker.

The confluence checker also uses information about never-stored and functional dependencies (see Section 5.6) to determine how many possible matchings (0, 1 or $*$) there are for each occurrence in a given rule. If there is only zero or one possible matchings, then the occurrence is trivially matching complete. These are very common cases in many programs. Otherwise if there are multiple possible matchings, it then checks for matching completeness as follows. Suppose that there are at least two matchings M_1 and M_2 for a given occurrence, then for matching completeness there are are two cases to consider: applying the **Simplify** or **Propagate** transition on M_1 *directly* deletes a constraint $c\#i \in M_2$ (e.g. Example 25); or executing the rule body *indirectly* deletes $c\#i \in M_2$ (e.g. Example 57).

For the first part, we check for direct deletion as follows. Let $(H_1 \backslash H_2)$ be the head of the rule, and let c be the active constraint, then there are two cases to consider. The first case is when the active constraint is deleted by the occurrence

(i.e. a member of H_2). To be matching complete, it must be that there can only ever be zero or one possible matchings (since c must is present in all matchings). To check this we use never-stored and functional dependency information to check if either one of $cons(H_1) \cup cons(H_2) - \{c\}$ is never-stored (therefore there cannot be any matchings) or the head $cons(H_1) \cup cons(H_2)$ is functionally determined by c (therefore there can only be one possible matching).

The other case where active c is not deleted by the occurrence is more complicated. We allow for four possible sub-cases:

1. the rule is a propagation rule, i.e. $H_2 = \emptyset$;
2. one of $cons(H_1) \cup cons(H_2) - c$ is never-stored;
3. the active constraint c functionally determines $cons(H_1) \cup cons(H_2)$;
4. for all $d \in cons(H_2)$ we have that d functionally determines $cons(H_1) \cup cons(H_2) - \{c\}$ and for all $d_1, d_2 \in (cons(H_1) \cup cons(H_2) - \{c\})$ we have that the predicate symbols of d_1 and d_2 are distinct.

Propagation rules can never directly delete constraints from any matching by definition, so they are safe. The second case and third cases are also trivially safe, since they imply there is only ever zero or one possible matchings. The forth case is more complicated. Suppose that for active $c\#i$ there is a constraint $d\#j$ and two matchings M_1 and M_2 such that $d\#j \in delete(M_1)$ and $d\#j \in delete(M_2)$. Then it must be that $M_1 - \{c\#i\} = M_2 - \{c\#i\}$, hence $M_1 = M_2$, otherwise d does not functionally determine the matching. Therefore, for all matchings M_1 and M_2 it must be that $delete(M_1) \cap delete(M_2) = \emptyset$.

Example 61 *Consider the following rule with three constraints in the head, and assume that all of these constraints are set semantic (at most one copy of each constraint).*

```
p \ q, r(X) <=> true.
```

Firstly note that the body cannot indirectly delete any constraint from any matching. The occurrence for `r(X)` *is matching complete because* `p` *and* `q` *are both (trivially) functionally determined by the active constraint (thanks to set semantics).*

The occurrence of `p` *is not matching complete because* `p` *nor* `q` *do not functionally determine* `r(X)`*. However, if we were to modify the rule to the following, then the occurrence of* `p` *is matching complete.*

```
p \ q(X), r(X) <=> true.
```

Now both `q(X)` *and* `r(X)` *functionally determine each other.* □

The second part of the matching completeness check tests if the body can indirectly delete a constraint from another matching. This information can be read from a call-graph of CHR constraints, and by examining the heads of rules to determine which CHR constraints can delete other CHR constraints.

Example 62 *For example, consider the program from Example 57.*

```
r1 @ p, q(X) ==> r(X).
r2 @ p, r(a) <=> true.
```

The call graph reveals that the body of rule `r1` *calls constraints of predicate symbol* `r`/1*. By examining the heads of the rules, we see that an active* `r`/1 *constraint may delete a* `p` *constraint. Therefore calling the body of rule* `r1` *may delete the active* `p` *constraint, hence the matching completeness check must fail.* □

If an occurrence fails matching completeness, then the confluence checker will try and prove matching independence, i.e. the choice of matching does not matter. Recall that matching independence is only applicable to occurrences where the active constraint is the only constraint deleted by the rule. A very simple matching independence test is to check if the free variables in the rule body are contained in the free variables of the active constraint. This was trivially true in Example 58, where the set of free variables in the body is empty. We can improve the matching completeness check by also allowing variables that are functionally determined by the active constraint.

Example 63 *Consider the following rule.*

```
r(Z), q(X,Y) \ p(X) <=> t(X,Y).
```

The occurrence for `p(X)` *is not matching independent because variable* `Y` *appears in the body, but not in the active constraint. If however there exists a functional dependency* $q(X,Y) :: \{X\} \rightsquigarrow \{X,Y\}$ *then the occurrence is matching independent.* □

Currently the HAL confluence checker assumes all occurrences are order independent by default, however the programmer can turn on order independence checking via a flag to the compiler. The order independence check is currently very weak, it involves finding all occurrences with more than one possible matching, and then checking if the rule body contains only built-in constraints (this is a surprisingly common case), or if the rule body is functionally determined by the active constraint. The programmer can also declare certain constraints and/or rules as "order independent", which narrows the checking to potential problem areas.

The order independence test could be improved by automatically checking some common CHR programming idioms. One such idiom is using constraints to accumulate some value, as was the case in Example 59. This can be generalised as follows. Suppose we have a rule of the form:

`p(`X`), p(`Y`) <=> p(`X *op* Y`).`

Where *op* is some binary operator that is symmetric and associative, e.g. addition $X + Y$ or set union $X \cup Y$, etc., then it does not matter which order goals of the form $\mathtt{p}(X_1), ..., \mathtt{p}(X_n)$ are called. Therefore any rule body consisting of a $\mathtt{p}(X)$ constraint is order independent.

6.4 Case Studies: Confluence Test

This section investigates the confluence of four CHR programs using our confluence checker. The programs are

- `ray` – a simple ray tracer;
- `bounds` – an extensible bounds propagation solver;
- `compiler` – Christian Holzbaur's bootstrapping CHR compiler; and
- `union` – union find algorithm implemented in CHRs (see [72]).

These particular programs are chosen because they were implemented before the confluence test and checker were invented. Therefore we can assume the programmer was not influenced by knowledge of the confluence test, which may affect design decisions, programming styles, etc.

6.4.1 Confluence of `ray`

The ray tracer is implemented in HAL [20] and has a total of 30 rules, 16 CHR constraints and 49 occurrences of CHR constraints on the left hand side of rules. The full source code (of a more updated version) is given in Appendix A.2. It can draw spheres, planes, multiple light-sources (of arbitrary colours) and handles simple shadowing. Primitive concepts, such as spheres and light rays, are given as input CHR constraints, while colours, shadows, etc. are calculated based on interactions between these constraints.

The confluence checker finds one matching problem (an occurrence that is neither matching complete nor order independent), and 3 order independence problems (occurrences that are matching complete but not necessarily order independent). The matching completeness problem appeared in:

```
lr1 @ intersection(IP,Id,D) \ light(LP,C) <=>
                                        light_ray(LP,IP,C,Id).
```

The `intersection` constraint is in fact an accumulator which keeps track of the nearest intersection with an object and the ray from the eye point.

```
int_near @ intersection(_,_,D1) \ intersection(_,_,D2) <=>
                                              D1 =< D2 | true.
```

This rule ensures that there is at most one possible `intersection` constraint in the store at once, however the functional dependency analysis in the compiler is too weak to detect this (because it currently does not take the guard into account). This can be fixed by the user asserting the functional dependency to the compiler, which currently is managed by adding the following rule.

```
int_fd @intersection(_,_,_) \ intersection(_,_,_) <=> true.
```

Of the five constraints that need to be verified as order independent, one is `intersection`, which is order independent since multiple copies will be reduced to the constraint with the minimum last argument. Other examples include constraints `color(C)` and `add_color(C)` whose behaviour was explained in Example 59. All five constraints had to be annotated as order independent.

6.4.2 Confluence of `bounds`

The bounds propagation solver is implemented in HAL and has a total of 83 rules, 37 CHR constraints and 113 occurrences. This version implements HAL's dynamic scheduling interface (which will be discussed in Chapter 8), as well as supporting ask constraints. A simpler bounds propagation solver is shown in Appendix A.1, however this version does not support dynamic scheduling nor ask constraints.

The confluence checker finds 4 matching problems, and 3 order independence problems. One of the matching problems indicated a bug (see below), the others are attributed to the weakness in the compiler's analysis. We only had to annotate one constraint as order independent.

The confluence analysis complained that the following rule is matching incomplete and non-independent when `kill(Id)` is active since there are (potentially) many possible matchings for the `delayed_goals` partner.

```
kill @ kill(Id), delayed_goals(Id,X,_,...,_) <=> true.
```

Here `delayed_goals(Id,X,_,...,_)` represents the delayed goals for bounds solver variable X. The code should be

```
kill1 @ kill(Id) \ delayed_goals(Id,X,_,...,_) <=> true.
kill2 @ kill(_) <=> true.
```

This highlights how a simple confluence analysis can be used to discover bugs.

The confluence analysis also complains the rules for bounds propagation themselves, e.g. the following rule handles bounds propagation for a `leq`/2 constraint.

```
leq @ leq(X,Y), bounds(X,LX,UX), bounds(Y,LY,UY) ==>
                                 bounds(X,LX,UY), bounds(Y,LX,UY).
```

The problem is that the constraint `bounds(`X`,`L`,`U`)` which stores the lower L and upper U bounds of variable X has complex self-interaction. Two `bounds` constraints for the same variable can interact using, for example,

```
b2b @ bounds(X,L1,U1), bounds(X,L2,U2) <=>
                                 bounds(X,max(L1,L2),min(U1,U2)).
```

Imagine an active bounds constraint visiting one of the occurrences for rule `leq`. The body of `leq` calls a new `bounds` which may delete the active constraint, and therefore the occurrences are indeed not matching complete.

Our program contains rules which are not matching complete and the confluence checker has identified them, however unlike before, matching incompleteness does not indicate a bug (the rules are still confluent). In this case confluence can be established by showing that the propagation rules, together with the other rules for the `bounds` constraint, are confluence under the theoretical semantics. Confluence under the refined semantics then follows because of Corollary 3. Unfortunately, proving confluence under the theoretical semantics is beyond the current implementation, so this is left for the programmer.

The confluence checker also reports potential order independence problems for the propagation rules. Technically order independence is irrelevant because these rules are not matching complete. Besides, if confluence under the theoretical semantics is established, order independence problems can be ignored.

6.4.3 Confluence of `compiler`

The bootstrapping compiler is implemented in SICStus Prolog (using the CHR library), including a total of 131 rules, 42 CHR constraints and 232 occurrences. It performs ad hoc analysis similar to that described in Chapter 5. After bootstrapping it has similar speed to the original compiler written in Prolog and produces more efficient code due to the additional analysis performed. During the bootstrap, when compiling itself the first time, the new code outperformed the old code (the SICStus Prolog CHR compiler, 1100 lines of Prolog) by a factor of five. This comparison is rather crude, measuring the costs and effects of the optimisations based on the additional analysis and the improved runtime system at once. Yet it demonstrates the practicality of the bootstrapping approach for CHRs and that CHRs as a general purpose programming language under the refined semantics can be used to write moderately large sized verifiable programs.

Bootstrapping CHRs as such aims at easier portability to further host languages and as an internal reality check for CHRs as a general purpose programming system. To the best of our knowledge, the bootstrapping compiler is the largest single CHR program written by hand. (Automatic rule generators for constraint propagation algorithms [8] can produce large CHR programs too, but from the point of the compiler their structure is rather homogeneous in comparison to the compiler's own code).

In order to use the confluence checker the `compiler` was (hastily) ported to HAL. This involved adding just enough HAL declarations to pass type, mode and CHR analysis (including confluence analysis).

The confluence checker finds 16 matching problems, and 45 order independence problems. 4 of the matching problems are removed by making functional dependencies explicit. The others are attributed to the weakness in the compiler's analysis. We had to annotate 18 constraints as order independent.

4 of the 16 matching problems turn out to be directly caused by undeclared functional dependencies, which were easily fixed. There were other matching problems, for example consider the rule.

```
merge_kvp @ key_value_pairs(Task,In), merge(Task,Key,Val) <=>
            key_value_pairs(Task,[Key-Val|In]).
```

This rule is a tricky loop, which accumulates (into the second argument of `key_value_pairs`) a list of $Key - Val$ pairs for every `merge` constraint associated with $Task$. For an active `key_value_pairs` constraint, there may be many possible matchings for the `merge` partner, hence the problem. The functional dependency on a `merge`($Task$,Key,Val) constraint was explicitly given (elsewhere in the program) as ($Task$,Key) determines Val. Therefore, an undeclared functional dependency is not at fault.

Technically, this rule is non-confluent, since the ordering of the final list does depend on the order of the matchings. However this is not a bug either, since the programmer is really representing an (un-ordered) set as a list (as is common practice). Hence, the final order of the list is irrelevant. To avoid the confluence warning, the rule can be re-written as a simpagation rule (which will try every matching), and an accumulator.

```
mkvp1 @ make_key_value_pairs(Task) \ merge(Task,Val) <=>
            key_value_pairs(Task,[Key-Val]).
mkvp2 @ make_key_value_pairs(_) <=> true.
key_value_pairs(Task,Ps1), key_value_pairs(Task,Ps2) <=>
                append(Ps1,Ps2,Ps3), key_value_pairs(Task,Ps3).
```

The main difference is the introduction of `make_key_value_pairs`($Task$) which must be called to generate the list. These rules pass the confluence test provided we also declare that `key_value_pairs` is order independent.

All other matching problems are caused by weakness in the current compiler's analysis. For example, consider the following program fragment which caused one of the warnings.

```
calls1 @ na_constraint(NA) \ calls(R,p(NA)) <=> calls(R,NA).
calls2 @ calls(R,p(NA)) <=> calls(R,any(NA)).
calls_uniq @ calls(R,any(_)) \ calls(R,_) <=> true.
```

The constraint `calls`(R,C) represents the relation that rule R calls C, which can be of the form `p`(NA), `any`(NA) or just NA. Initially, all `calls/2` are of the form `calls`(R,`p`(NA)), but are later refined to either `calls`(R,NA) or `calls`(R,`any`(NA)) depending on whether NA represents the name/arity of a CHR constraint, or a predicate (in which case we assume it can call any CHR constraint).

For active `na_constraint`(NA), there may be many possible matchings for `calls`(R,`p`(NA)). The body of the rule calls another `calls` constraint, which in turn deletes other `calls` constraint by the third rule. This leads to the problem of potential matching incompleteness, since the body calls constraints `calls` which may delete matching partners, also `calls` constraints.

A more careful analysis reveals that for active `na_constraint(`NA`)` there cannot be any possible matching for a `calls(`R`,p(`NA`))` constraint. The reason is that the second rule will always delete an active constraint of the form `calls(`R`,p(`NA`))`, hence by the time `na_constraint(`NA`)` is active, there are no such constraints in the store. If there are no matching partners, this cannot be a universal search. The confluence checker doesn't detect this because its current analysis does not analyse individual arguments of any matching/called constraints.

To overcome this weakness, the above program fragment can be rewritten to split the `calls` constraint into two more specialised constraints, e.g. `calls_p` and `calls`. Hence our code fragment becomes.

```
na_constraint(NA) \ calls_p(R,NA) <=> calls(R,NA).
calls_p(R,NA) <=> calls(R,any(NA)).
calls_uniq @ calls(R,any(_)) \ calls(R,_) <=> true.
```

Now the checker can determine that `calls_p` will never be stored after it is active, and that an active `na_constraint` will have no matching partners, hence no confluence warning is reported.

Unlike the `ray` and `bounds` examples, there were a relatively large number of rules/constraints that need to be verified and annotated order independent. This is because of the design of the compiler, which performs various analysis, the results of which affect other parts of the compilation process.

There are some easy cases to verify order independence. For example, `key_value_pairs` constraint needs to be verified order independent, which it is because its usage does not depend on the order of the resulting list. There are many other similar examples where the constraints are order independent because they implement some sort of accumulator.

There are some cases of where order independence is much harder to verify. For example, consider the following rule

```
not_set @ set(NA1,no), interact(NA1,NA2) ==> set(NA2,no).
```

The constraint `set(`NA`,`YN`)` represents whether or not constraint NA has set semantics. Constraint `interact(`$NA1$`,`$NA2$`)` represents if $NA1$ and $NA2$ are both present in the head of some rule. The rule states that if $NA1$ does not have set semantics, and $NA1$ interacts with $NA2$, then $NA2$ also does not have set semantics. For active `set(`$NA1$`,no)` there can be many possible matchings for `interact(`$NA1$`,`$NA2$`)`, hence we want to ensure that the order `set(`$NA2$`,no)` constraints are generated does not matter.

The first rule for `set` constraint is

```
first_value @ set(NA,_) \ set(NA,_) <=> true.
```

Generally, such a rule would guarantee `set(`NA`,_)` constraint is not order independent, e.g. consider executing `set(f/1,yes), set(f/1,no)` and in the reverse order.

Despite this, the rule `not_set` is order independent for active `set` constraint. The reason it that the rule generates constraints of the form `set(`$NA2$`,no)`, where the second argument is always fixed to `no`. Also, two `set` constraints with different first arguments are not considered by the `first_value` rule. Therefore, this restricted form of `set` constraints is indeed order independent in their limited usage. A majority of the other rules were verified order independent by similar reasoning.

It also became clear in our exploration of the bootstrapping CHR compiler using the confluence checker that had the confluence checker been available during the writing of the bootstrapping compiler it could well have been written in a substantially different way in order to avoid some of the problems reported by the checker.

6.4.4 Confluence of `union`

The naive union find algorithm was originally implemented for the K.U.Leuven CHR system [71], but now has been ported to HAL. It consists of 7 rules, 6 CHR constraints and 11 occurrences.

The confluence checker finds 2 matching completeness problems. Both of these problems exposed implicit assumptions about functional dependencies and never-stored constraints.

For example, the confluence checker complains that the following rule is not matching complete (for any occurrence).

```
link @ link(A,B), root(A), root(B) <=> arrow(B,A), root(A).
```

This is solved by the addition of two rules: the first declares `root`/1 to be set semantic and the second explicitly deletes unused `link`/2 constraints (the programmer was implicitly assuming that `link`/2 will always fire the `link` rule, however the confluence checker cannot be expected to detect this). Thus the result is

```
root(A) \ root(A) <=> true.
link @ link(A,B), root(A), root(B) <=> arrow(B,A), root(A).
link(_,_) <=> true.
```

which passes the confluence checker.

6.5 Summary

The refined operational semantics for Constraint Handling Rules provides a powerful and expressive language, ideal for applications such as compilers, since fixpoint computations and simple database operations are straightforward to program. The disadvantage of CHRs over other possible languages is that CHRs do not have a fully deterministic operational semantics, so to counter this problem

the programmer usually aims to write confluent CHRs programs. Unfortunately, the Abdennadher confluence test is too strong for the refined operational semantics, so we have presented a novel static confluence checker based on information obtained from standard CHR program analysis.

The confluence test identifies four properties that if satisfied, guarantees confluence under the refined semantics. These are: termination, trivial wakeup policy, matching completeness or independence, and order independence. The termination requirement is solely left for the programmer, and order independence typically requires help from the programmer (in current implementations). Groundness and matching completeness/independence can be checked automatically in modern CHR compilers.

We implemented a confluence checker for HAL CHR based on the confluence test, and evaluated the checker on four CHR programs: a simple ray tracer, a bounds propagation solver, a bootstrapping CHR compiler and the CHR version of the union find algorithm. By far testing for matching completeness or independence is the most useful, since the majority of all occurrences the case studies were either matching complete, matching independent, or in one instance indicated a bug. The exceptions occur when the programmer is relying on confluence under the theoretical semantics, as with some of the rules in the `bounds` example.

Matching completeness exposes the programmer's implicit assumptions about functional dependencies, as was shown by the case studies. This is good because it encourages the programmer to declare functional dependencies explicitly, which has other benefits, such as optimisation (this will be discussed in the next chapter).

Order independence remains a difficult property to test for, hence the confluence checker usually requires help in the form of user annotations. In order not to overwhelm the programmer, the compiler can try its best to exclude as many occurrences as possible, for example, when the body of the rule contains only built-in constraints, and when the body is functionally determined by the active constraint. A more sophisticated compiler can also try to exclude some other cases, e.g., when the body of a rule is calling constraints that just accumulate some value, etc.

Unfortunately the confluence test performs poorly when the programmer writes code with complex interactions, e.g. the `bounds` constraint and propagators. In these cases the programmer is usually relying on confluence under the theoretical operational semantics, which our current implementation cannot detect (although this may be future work). In such cases the programmer can ignore, or disable the confluence checker, or reformulate the program so that it complies with the confluence test in Definition 46.

Chapter 7

Optimisation

7.1 Introduction

In this chapter we discuss how to improve the basic compilation of CHRs by using additional information derived either from declarations provided by the user or from the analysis of the CHRs themselves. The major improvements we discuss are:

- finding a good order for calculating joins and scheduling guards during the join;
- general index structures which are specialised for the particular joins required in the CHR execution;
- continuation optimisation, where we use matching information from rules earlier in the execution to avoid matching later rules; and
- removal or postponement of overhead/initialisation

We illustrate the advantages of the various optimisations experimentally on a number of example programs in the HAL implementation of CHRs. We also discuss how the extra information required by HAL in defining CHRs (that is, type, mode and determinism information) is used to improve the execution.

The remainder of this chapter is organised as follows. We divide the optimisations into two main groups. Local optimisation in Section 7.2 looks at how to optimise each individual occurrence in isolation. Global optimisation in Section 7.3 we show how to further optimise the program by taking into account the context of each occurrence with respect to other occurrences. In Section 7.4 we give our experimental results illustrating the advantages of the optimised compilation. Finally, we conclude.

7.2 Local Optimisation

The bulk of the execution time for any given rule ($H_1 \setminus H_2 \iff g \mid C$) is spent in determining which constraints H_1' and H_2' from the store match against the head of the rule. Under the basic compilation, this is the role of the *occurrence*, *join-loop* and *call-body* predicates. This section is concerned with local optimisations which are applicable to the set of predicates generated per occurrence. We will show how different kinds of compile-time information can be used to improve the resulting code in the HAL version of CHRs.

7.2.1 Overhead Removal

We very loosely define *overhead* to be any part of the generated code which is not directly involved in the algorithm defined by the original CHR program, e.g. history checks, liveness checks, etc. Under the basic compilation, the generated code is usually riddled with various forms of overhead, which can penalise runtime performance. In this section we look at removing some of this overhead under favourable conditions. This is particularly important for simple CHR programs, because the amount of overhead is generally disproportionate to the actual work the program does. For example, the compiled version of `gcd` in Figure 4.6 is much larger than the hand written version of Example 30 mainly because of the amount of overhead.

Universal vs. existential Searches

Under the basic compilation, the collection of join-loop predicates are designed to search through all possible matches for a given occurrence. This is fine for some rules, e.g. propagation rules, where all possible matches are actually required. However only one match is required for simplification rules, since once the rule fires the active constraint is deleted (hence no more matches are needed). We now differentiate between two kinds of searches for matchings: A *universal* search which iterates over all possible matchings, and an *existential* search which looks for the first possible match. Under the basic compilation, we are essentially approximating existential searches using universal searches, and this adds unnecessary overhead.

We can generalise universal/existential searches even further: let ($H_1 \setminus H_2 \iff g \mid C$) be a rule. The partner search uses universal search behaviour, up to and including the first constraint in the join which appears in H_2 (the part of the head deleted by the rule). From then on the search is existential. If the constraint has a functional dependency that ensures that there can be only one matching partner, we can replace universal search by existential search.

For existential searches we can use HAL nondeterministic search inside the condition-part of an if-then-else to find (and then commit to) a single matching partner. The simplified join-loop predicate for existential searches is shown in Figure 7.1. The interface in line (1) is exactly the same as under the basic

```
p_1_join_loop(Ls,Id_1,...,Id_i,X_1,...,X_n) :-                (1)
    ( member(Ls,C # Id),                                      (2)
      C = q(A,B,C),                                           (3)
      alive(Id),                                              (4)
      Id \= Id_1,                                             (5)
      ⋮
      Id \= Id_i ->                                           (6)
        <find-matches-and-call-body>                          (7)
    ;   true
    ).
```

Figure 7.1: Simplified join-loop predicate based on nondeterministic search

compilation. Line (2) is the call to `member`/2, which nondeterministically searches for a potential matching partner. Lines (3)-(7) exactly correspond to lines (3)-(7) in Figure 4.3 under the basic compilation. We have completely removed the overhead of the recursive call to `p_1_join_loop` and the associated liveness tests for $\texttt{Id}_1,\ldots,\texttt{Id}_i$ (see lines (9)-(12) from Figure 4.3). This is because after line (7) we have either found no matchings, or found one matching and successfully fired the rule.

Example 64 *Consider the compilation of the first occurrence of the* ***bounds/3*** *constraint in the following rule (the fourth occurrence overall in the program in Appendix A.1)*

```
intersect @ bounds(X,L1,U1)_4, bounds(X,L2,U2)_5 <=>
                          bounds(X,max(L1,L2),min(U1,U2)).
```

Since the active constraint is deleted the entire search is existential. The compilation produces the code in Figure 7.2. Predicate ***bounds_4_join_loop*** *iterates nondeterministically through the* ***bounds***/3 *constraints in the store, and then commits to the first matching.* □

The idea of replacing universal with existential searches first appeared in [47][1] and was implemented in the SICStus CHR compiler. The older version of the optimisation is very similar to ours, except that it did not take into account functional dependency information (which is not available to the SICStus compiler). It was also shown that the optimisation is beneficial for the SICStus CHR system.

Removal of propagation histories

Even with a clever implementation, the propagation history is still a relatively expensive data structure to maintain. Under the basic compilation we assumed a

[1] In [47] a *deterministic recursive loop* is synonymous to a universal search, and *nondeterministic backtracking search* is synonymous to an existential search.

```
bounds_4(Id,X,L1,U1) :-
    get_iterator(Ls),
    bounds_4_join_loop(Ls,Id,L1,U1),
    ( alive(Id) ->
        bounds_5(Id,X,L1,U1)
    ;   true
    ).

bounds_4_join_loop(Ls,Id,L1,U1) :-
    ( member(Ls,C # Id1),                  %% nondet search
      C = bounds(X,L2,U2),                 %% match?
      alive(Id1),                          %% alive?
      Id1 \= Id                            %% different?
    ->                                     %% commit
        delete(Id),                        %% delete active
        delete(Id1),                       %% delete match
        bounds(X,min(L1,L2),max(U1,U2))    %% Body
    ;   true
    ).
```

Figure 7.2: Existential search code for the fourth occurrence of a `bounds`/3 constraint

single monolithic propagation history for all rules (as is the case with the specification of the refined operational semantics). We can improve on this slightly by generating a propagation history on a per propagation rule basis, i.e. generate a specialised `check_history_`r`(Entry)` per propagation rule r. While this is of some benefit, checking the history remains a costly operation. Fortunately, there are cases where a propagation history can be removed altogether, thanks mainly to the way the iteration works.

Let `p(`$\bar{x}$`)` be a CHR constraint that is never woken up during a **Solve** transition at runtime. This is the case whenever all of $\bar{x}$ are ground (see Definition 11). We now consider how constraint `p(`$\bar{x}$`)` becomes active, which must be by either an **Activate** or **Reactivate** transition. Since **Reactivate** is only applicable to numbered constraints, which have been woken up by **Solve**, we can exclude this case. Thus `p(`$\bar{x}$`)` is active only once during a derivation via the **Activate** transition.

Let $(r \ @ \ h_1, ..., h_n \Longrightarrow g \mid B)$ be a propagation rule in program P. Suppose that $h_1, ..., h_n$ contains the k^{th} occurrence for CHR constraint c in P. We can omit the propagation history check for occurrence k if the following holds for all possible matchings $c\#i \in M$ for $h_1, ..., h_n$.

1. M is ground/fixed at runtime; and

2. For all $c'\#i' \in M$ such that $i' \neq i$, it is not possible that constraint c called constraint c' before occurrence k.

Suppose that $c =$ `p(`$\bar{x}$`)`. These conditions ensure that once `p(`$\bar{x}$`)` becomes active (only possible by **Activate** thanks to the first condition) no new potential matches for rule $h_1, ..., h_n$ are created before active `p(`$\bar{x}$`)` reaches occurrence k. We give a semi-formal argument of why these conditions remove the need for a propagation history check.

The active constraint $p(\bar{x})\#i : k$ is implemented by the call `p_`k`(`i`,`$\bar{x}$`)` under the basic compilation. At this point a global iterator `Ls`, which is essentially a copy of the CHR store, is obtained. A universal search over `Ls` is performed, i.e. iterating through all possible matches in `Ls`. Assume that there are $M_0, ..., M_m$ matches in `Ls`. Thanks to iteration, when `p(`$\bar{x}$`)` is active we only consider each $M \in \{M_0, ..., M_m\}$ once before trying other matches, hence each M fires the rule at most once during the entire iteration for occurrence k. We can extend this even further: Condition 1 ensures that `p(`$\bar{x}$`)` is only ever active once, therefore each M fires the rule at most once for active `p(`$\bar{x}$`)` throughout the entire derivation. This is exactly the correct behaviour (match M firing the rule at most once), with or without an explicit propagation history.

So far we have only solved half the problem: when other constraints in M are active, they may fire the rule r with the same match M as well. This is where Condition 2 is useful. Assume that M may fire rule r when `p(`$\bar{x}$`)` is active (as in the previous paragraph). Let `q(`$\bar{y}$`)` be another constraint in M that may fire rule r with the same match M when `q(`$\bar{y}$`)` is active. Usually the propagation history will prevent this from happening. Without loss of generality, assume that `q(`$\bar{y}$`)` became active after `p(`$\bar{x}$`)`. Then `p(`$\bar{x}$`)` must have (directly or indirectly) called `q(`$\bar{y}$`)`. Furthermore, `p(`$\bar{x}$`)` must have called `q(`$\bar{y}$`)` before occurrence k of `p` (otherwise `p(`$\bar{x}$`)` could not have fired rule r with M, since `q(`$\bar{y}$`)` was not yet in the store). Therefore we have violated Condition 2.

Here is a simple example of violation of Condition 2, which also demonstrates the need for a propagation history.

Example 65 *Consider the following program where the second rule violates Condition 2 (an active* `p` *may call* `q` *before the propagation rule).*

```
r1 @ p \ s <=> q.
r2 @ p, q  ==> r.
```

Under the HAL CHR implementation, without an explicit propagation history for rule `r2`*, a call to* `p` *incorrectly calls* `r` *twice (if there is a matching* `s` *for rule* `r1` *in the store). In this situation the HAL CHR compiler will generate a propagation history. If, however, rule* `r1` *were a simplification rule, e.g.,*

```
r1 @ p, s <=> q.
```

Then rule `r2` *does not require a propagation history. Although it is possible that an active* `p` *calls* `q`*, the active* `p` *is deleted in the process, and therefore it is safe to remove the propagation history for rule* `r2`*.* □

The current HAL implementation removes propagation histories whenever possible. Groundness (Condition 1) information comes from the mode declarations for CHR constraints, and Condition 2 can be directly read off a CHR call graph.

Other improvements

There are other simple improvements over the basic compilation. These include

1. Removal of the call to `delay`/3 if a CHR constraint does not extend any solvers, e.g. `gcd`/1 does not extend any solver, so a call to `delay`/3 is not required;

2. Removal of constraint identifier difference tests, i.e. the tests `Id \= Id`$_1$ etc. from join-loop predicates (see Figure 4.3), if the compiler can prove that the constraints must be different, e.g. if they have different functor/arities;

3. Removal of redundant liveness tests, i.e. `alive(Id)` etc. from join-loop predicates, if the compiler can prove that the test will always succeed. This is possible if the body of a rule cannot cause the deletion of a constraint from any of the iterators;

4. Remove calls to `delete`/1 in call-body predicates if the body of the rule calls `fail` before any CHR constraint;

5. Reducing the number of arguments to join-loop and body-call predicates to the actual arguments that are used (e.g. by the guard or body).

6. Inlining predicates, e.g. inlining the call-body predicates, etc.

Another benefit is that the size of the resulting code is reduced, which can make the code more readable, and make compilation faster.

7.2.2 Join Ordering and Early Guard Scheduling

The head of a rule together with the guard defines a multi-way join with selections (the guard) that could be processed in many possible ways, starting from the active constraint. Under the basic compilation of Chapter 4, the order at which partner constraints are matched (i.e. the caller/callee relationship between join-loop predicates) is unspecified. In most older CHR compilers, e.g. the original SICStus CHR compiler, the calculation of partner constraints is performed in textual order. In both cases the guards are evaluated once all partners have been identified, i.e. inside the call-body predicate. By testing guards as partners are matched, i.e. inside join-loop predicates, and finding a better order for matching partners, we can dramatically improve the run-time performance of CHR programs.

The problem of join ordering has been studied before, e.g. in the context of production systems using the TREAT matching algorithm [63]. Join ordering for CHRs is more complicated because the compiler needs to consider more complex guards. A similar problem has also been extensively addressed in the database literature, however most of this work is not applicable since in the database context they assume the existence of information on cardinality of relations (number of stored constraints) and selectivity of various attributes. Since we are dealing with a programming language we have no access to such information, nor reasonable approximations. Another important difference is that, often, we are only looking for the first possible join partner (i.e. an existential search), rather than all.

Since we have no cardinality or selectivity information we will select a join ordering by using the number of unknown attributes in the join to estimate its cost. Functional dependencies are used to improve this estimate, by eliminating unknown attributes from consideration that are functionally defined by known attributes.

The function fdclose($Fixed$,FDs) closes a set of fixed variables $Fixed$ under the finite set of functional dependencies FDs. Note that for *full* functional dependencies fdclose($Fixed$,FDs) will either do nothing or add all variables in FDs to the result.

We assume an initial set $Fixed = vars(c)$ of known variables (where c is the active constraint), together with the set of (as yet unprocessed) partner constraints and guards. We also assume that all partner constraints and guards have been normalised (see Section 4.2.2). The algorithm measure shown in Figure 7.4, takes as inputs the set $Fixed$, the sequence $Partners$ of normalised partner constraints in a particular order, the set FDs of functional dependencies and the set $Guards$ of guards, and returns the triple $(Measure, Goal, Lookups)$.

$Measure$ is an ordered pair representing the cost of the join for the particular order given by the n partner constraints in $Partners$. It is made up of the weighted sum $(n-1)w_1 + (n-2)w_2 + \cdots + 1w_{n-1}$ of the costs w_i for each individual join with a partner constraint. The weighting encourages the cheapest joins to be earliest.

The cost of joining the i^{th} partner constraint to pre-join expression (the join of the active constraint plus the first $(i-1)$ partners), w_i, is defined as the pair (u, f): u is the number of arguments in the new partner which are unfixed before the join; and f is the negative of the number of arguments which are fixed in the pre-join expression. The motivation for this costing is based on the worst case size of the join, assuming each argument ranges over a domain of the same size s. In this case the number of join partners (tuples) in the partner constraint for each set of values for the pre-join expression is s^u, and there are s^{m-f} tuples in the pre-join expression (where m is the total number of variables in the pre-join expression). The total number of tuples after the i^{th} partner is joined are thus s^{m-f+u}. The numbers hence represent the exponents of the join size, a kind of "degrees of freedom" measurement. The sum of the first components u gives the

Condition	*Guard*	*Selectivity*
$fixed(X)$ and $fixed(Y)$	$X = Y$	1.0
$\neg fixed(X)$ or $\neg fixed(Y)$	$X = Y$	0.0
$fixed(X)$	$X = f(...)$	1.0
$\neg fixed(X)$	$X = f(...)$	0.0
`p is failure`	$p(...)$	∞
$fixed(X)$ and `(p is semidet)`	$p(..., X, ...)$	0.25 per X
$\neg fixed(X)$ or `(p is det)`	$p(..., X, ...)$	0.0 per X

Figure 7.3: Example selectivity approximations of various guards.

total size of the join and the role of the second component is to prefer orderings which keep the intermediate results smaller.

We also take into account the selectivity of the guards we can schedule directly after the new partner. The role of selectivity is to encourage the early scheduling of guards which are likely to fail. This is achieved via the *selectivity*(*Guards*) function which returns the sum of the selectivities of the *Guards*. The *selectivity* function for the HAL CHR compiler is shown in Figure 7.3, where the selectivity of an equation guard $X = Y$ is 1.0 provided X and Y are both fixed, otherwise the selectivity is 0.0. An equation with both X and Y fixed immediately eliminates one degree of freedom (reduces the number of tuples by $1/s$), hence the selectivity of 1.0. When one variable is not fixed, the guard never removes any answers, hence the selectivity of 0.0. The case where the guard is a function call, i.e. $X = f(...)$, is similar: selectivity is 1.0 if X is fixed, or 0.0 otherwise. The last three cases handle arguments to predicate calls (including built-in constraints) and function calls in the guard. Here, we guess the selectivity based on the determinism of the predicate. If the determinism is `failure`, then this indicates the strongest possible selectivity (nothing is ever selected), hence ∞ (or a very large number) is returned.[2] Otherwise if the determinism is `semidet`, our approximation of the selectivity is 0.25 per fixed argument (as motivation, the constraint $X > Y$ with two fixed arguments can be considered to remove 0.5 degrees of freedom). Finally, if the determinism is `det`, which is weakest possible selectivity (everything is selected), hence 0.0 is returned.

Example 66 *Consider the guard* `X = 2 ^ Y` *where* X *and* Y *are integers (hence function* `(2^)` *is* `semidet`*) with* X *and* Y *fixed. In HAL, this guard will be assigned a selectivity of* 1.25 *(*1.0 *for* X *being fixed plus* 0.25 *for* Y *being a fixed argument to a* `semidet` *function).*

Notice how the assigned selectivity is always just an approximation. A better value could always be chosen if we take into account more information. For example, if Y *was known to be always positive (e.g. the guard* $Y \geq 2$ *was scheduled*

[2] If the determinism is `failure`, then the occurrence will be removed by the *success continuation optimisation*, which is described later in Section 7.3.1.

```
measure(Fixed,Partners,FDs,Guards)
    Lookups := ∅
    score := (0, 0)
    sum := (0, 0)
    Goal := tell_to_asks(Guards)
    Guards := Guards \ Goal
    while true
        if Partners = ∅
            return (score, Goal ++ tell_to_asks(Guards), Lookups)
        let Partners ≡ [p(x̄)|Partners1]
        Partners := Partners1
        FD_p := {p(x̄) :: fd ∈ FDs}
        Fixed_p := fdclose(Fixed, FD_p)
        f̄ := x̄ ∩ Fixed_p
        Fixed := Fixed ∪ x̄
        GsEarly := schedule_guards(Fixed, Guards)
        cost := (max(|x̄ \ f̄| − selectivity(GsEarly), 0), −|f̄| − selectivity(GsEarly))
        score := score + sum + cost
        sum := sum + cost
        Lookups := Lookups ∪ {p((x_i ∈ f̄ ? x_i : _) | x_i ∈ x̄)}
        Goal := Goal ++ [p(x̄)] ++ GsEarly
        Guards := Guards \ GsEarly
    endwhile
    return (score, Goal, Lookups)

schedule_guards(Fixed,Guards)
    GsEarly := []
    repeat
        Guards0 := Guards
        foreach g ∈ Guards
            if invars(g) ⊆ Fixed
                GsEarly := GsEarly ++ [g]
                Fixed := Fixed ∪ outvars(g)
                Guards := Guards \ {g}
    until Guards0 = Guards
    return tell_to_asks(GsEarly)
```

Figure 7.4: Algorithm for evaluating join ordering

earlier in the join), then the function `(2^)` *is effectively total, hence a better approximation for the selectivity is* 1.0. □

Goal gives the ordering of partner constraints and guards (with guards scheduled as early as possible). Finally, *Lookups* gives the lookups (or queries). Queries will be made from partner constraints, where a variable name indicates a fixed

value, and an underscore (_) indicates an unfixed value. For example, lookup `p(X,_,Y,Z,_)` indicates a search for `p/5` constraints with a given value in the first, third, and fourth argument positions.

The function schedule_guards($Fixed$,$Guards$) returns which guards in $Guards$ can be scheduled given the fixed set of variables $Fixed$. Here we see the usefulness of mode information which allows us to schedule guards as early as possible. For simplicity, we treat mode information in the form of two functions: $invars$ and $outvars$ which return the set of input and output arguments of a guard procedure. We also assume that each guard has exactly one mode (it is straightforward to extend the approach to multiple modes[3] and more complex instantiations). The schedule_guards keeps adding guards to its output argument while they can be scheduled. We assume the function tell_to_asks converts any tell constraints in the scheduled guards to the corresponding ask constraint version as per described in Section 4.5.

The function measure works as follows: beginning from an empty goal, we first schedule all possible guards. We then schedule each of the partner constraints $p(\bar{x})$ in $Partners$ in the order given, by determining the number of fixed ($\bar{f}$) and unfixed ($\bar{x} \setminus \bar{f}$) variables in the partner, and the selectivity of any guards that can be scheduled immediately afterwards. With this we the calculate *cost* pair for the join which is added into the *score*. The *Goal* is updated to add the join $p(\bar{x})$ followed by the guards that can be scheduled after it. When all partner joins are calculated the function returns.

Example 67 *Consider the compilation of the (normalised) rule:*

```
p(X,Y), q(Y1,Z,T,U), flag, r(X1,X2,U1) \ s(W) <=>
    U = U1, X = X1, X1 = X2, Y = Y1, W = U + 1, linear(Z)
        | p(Z,W).
```

for active constraint `p(X,Y)` *and* $Fixed = \{X, Y\}$. *The individual costs calculated for each join in the left-to-right partner order illustrated in the rule are* $(2.75, -1.25)$, $(0, 0)$, $(0, -3)$, $(0, -1)$ *giving a total cost of* $(10, -12)$ *together with goal*

```
Y = Y1, X = X1, X1 = X2, q(Y1,Z,T,U), W = U + 1, U = U1, linear(Z),
flag, r(X1,X2,U1), s(W)
```

and lookups `q(Y1,_,_,_)`, `flag`, `r(X1,X2,U1)`, `s(W)`. *The best order has total cost* $(4.75, -8.25)$ *resulting in goal*

```
Y = Y1, X = X1, X1 = X2, flag, r(X1,X2,U1), U = U1, W = U + 1, s(W),
q(Y1,Z,T,U), linear(Z)
```

and lookups `flag`, `r(X1,X2,_)`, `s(W)`, `q(Y1,_,_,U)`.

For active constraint `q(Y1,Z,T,U)`, *the best order has total cost* $(2, -9)$ *resulting in goal*

[3]The HAL CHR compiler supports guards with multiple modes.

```
Y = Y1, U = U1, W = U + 1, linear(Z), s(W), flag, p(X,Y), X = X1,
X1 = X2, r(X1,X2,U1)
```

and lookups `s(W)`, `flag`, `p(_,Y)`, `r(X1,X2,U1)`. □

Aliased variables introduced because of normalisation can be removed by applying the appropriate substitution. For example, we can simplify the last goal in Example 67 to

```
W = U + 1, linear(Z), s(W), flag, p(X,Y), r(X,X,U)
```

with (more specialised) lookups `s(W)`, `flag`, `p(_,Y)`, `r(X,X,U)`. We call this process *denormalisation*, which helps reduce the number of arguments for join-loop and call-body predicates. The HAL CHR compiler denormalises all goals after join ordering.

For rules with large heads where examining all permutations is too expensive we can instead use heuristics to search for a permutation of the partners that is likely to be cost effective. The current HAL CHR compiler uses a hybrid approach. Assume that there are m partners to schedule, then the first $m - n$ partners are scheduled greedily, for some user configurable number n. The remaining n partners are scheduled by brute force by trying all permutations.[4] Note that since m is usually at most 3 in practice, this is rarely an issue and the most optimal permutation is chosen.

7.2.3 Index Selection

Once join orderings have been determined, we must select a set of indexes for each constraint in the store. Under the basic compilation, where no indexing was used, a call to `get_iterator` always returns a list iterator which contains every CHR constraint in the store. This is clearly inefficient, since (in general) very few of the constraints in the iterator actually represent valid matches. The aim of indexing is to specialise calls to `get_iterator` such that the iterator only contains constraints of a particular form, namely the form of a given lookup. For example, an iterator from an index for the lookup `p(_,X)` will only contain constraints of the form `p(_,X)`. The advantage is that the amount of iteration is reduced when calculating the join, which can decrease the runtime complexity of the program. In this section we examine how to chose suitable indexes given a set of lookups, and how these indexes are actually implemented in HAL.

The current Prolog CHR implementations[5] use only two index mechanisms: Constraints for a given functor/arity are grouped, and variables shared between heads in a rule *index* the constraint store because matching constraints must

[4]Note that is better to use brute force before the greedy algorithm, however this is not currently implemented.

[5]The exception is the Schrijvers' CHR implementation, which uses general indexing for ground constraints based on [45] (an earlier version of this work).

correspondingly share a (attributed) variable (see Section 2.3.3). In the HAL CHR compiler, we put extra emphasis on more general indexes.

For simplicity, we will assume that the constraints in the store (and hence also in the indexes) are always ground, because variable bindings will change the correct position of data in the index. Later, in Chapter 8, we will overcome this restriction and allow indexes on non-ground data as well. As this requires additional non-trivial help from the underlying solver, we postpone consideration until later.

The data structure used to implement an index (or *index structure* from now on) consumes memory, so one of our aims is to generate as few index structures as possible. To achieve this, usually the compiler will perform *lookup reduction*: given a set of lookups for constraint `p/`k we reduce the number of lookups (and hence the number of indexes) by using information about properties of `p/`k. The rules for lookup reduction used by HAL are:

- *lookup generalisation*: rather than build specialised indexes for lookups that share variables we simply use more general indexes. Thus, we replace any lookup $\texttt{p}(v_1, \ldots, v_k)$ where v_i and v_j are the same variable by a lookup $\texttt{p}(v_1, \ldots, v_{j-1}, v'_j, v_{j+1}, \ldots, v_k)$ where v'_j is a new variable. Of course, we must add an extra guard $v_i = v_j$ for rules where we use generalised lookups. For example, the lookup `r(X,X,U)` can use the lookup for `r(X,Y,U)`, followed by the guard `X = Y`.

- *functional dependency reduction*: we can use functional dependencies to reduce the requirement for indexes. We can replace any lookup $\texttt{p}(v_1, \ldots, v_k)$ where there is a functional dependency $p(x_1, \ldots, x_k) :: \{x_{i_1}, \ldots, x_{i_m}\} \rightsquigarrow \{x_1, \ldots, x_k\}$ and $v_{i_1}, \ldots, v_{i_m}$ are fixed variables (i.e. not `_`) by an equivalent lookup where each (non-fixed) $v \notin v_{i_1}, \ldots, v_{i_m}$ is replaced by '`_`'. For example, consider the constraint `bounds`/3 from the `bounds` program (see Appendix A.1). Given functional dependency $bounds(X, L, U) :: \{X\} \rightsquigarrow \{X, L, U\}$, the lookup `bounds(X,L,_)` can be replaced by `bounds(X,_,_)`.

After lookup reduction, the compiler generates an index structure for every remaining lookup. The rest of this section is concerned with choosing the right index for a given lookup.

Each index for $\texttt{p}(v_1, \ldots, v_k)$, where say the fixed variables are $v_{i_1}, \ldots, v_{i_m}$, needs to support the operations shown in Figure 7.5. for initialising a new index, inserting and deleting constraints from the index and returning an iterator for a given lookup. The name *index* is a unique string used to identify the index (recall that multiple indexes are possible for the same constraint). Unlike basic compilation, where iterators are always lists of identified constraints, the type of *iterator* will depend on the data-structure used to implement the *index*. We also allow a call to `p_`*index*`_get_iterator` to fail, which indicates an empty iterator. We are assuming that the type of a constraint identifier is `chr_id`, which is required for insertion and deletion. In HAL, indexes are stored in global variables,

```
:- pred p_index_init.
:- mode p_index_init is det.

:- pred p_index_insert(arg1, ..., argk, chr_id).
:- mode p_index_insert(in, ..., in, in) is det.

:- pred p_index_delete(arg1, ..., argk, chr_id).
:- mode p_index_delete(in, ..., in, in) is det.

:- pred p_index_get_iterator(argi1, ..., argim, iterator).
:- mode p_index_get_iterator(in, ..., in, out) is semidet.
```

Figure 7.5: Operations that need to be supported for each index.

Index Structure	*Iterator Type*
`yesno`	T
`tree`	T
`tree*`	`list(`T`)`
`hashtable`	T
`hashtable*`	`list(`T`)`
`list`	`list(`T`)`

Figure 7.6: Supported index structures and corresponding iterator types

which are destructively updated for initialisation, deletions and insertions. The compiler generates code for the predicates `p_insert` and `p_delete` which insert and delete the constraint `p` from each of the indexes in which it is involved.

The HAL CHR compiler supports several kinds of index structures as shown with their corresponding iterator types in Figure 7.6. Here T represents the type of identified CHR constraints, i.e. the type of the terms `C # Id`. Unlike basic compilation, where all iterators where of the type $list(T)$, some index structures (i.e. `yesno`, `tree`, etc.) always return a singleton iterator, hence the type T. The main advantage is that existential searches can be *determinised* by replacing the (nondeterministic) call to `member(Ls,C#Id)` with the (deterministic) deconstruction `(Ls = C#Id)`. In HAL, deterministic predicates are usually faster than a similar nondeterministic alternative because of less overhead.

By far the simplest index structure is `yesno`, which can have two states: a `no` state (meaning nothing is currently stored) or a `yes(C)` state, where C is the only (numbered) constraint currently in the store. The `yesno` index is an example of an index structure that always returns a singleton iterator, i.e. when the state is `yes(C)`. The compiler will generate a `yesno` index structure whenever

it detects that it is not possible for multiple[6] `p(`$\bar{x}$`)` constraints to exist in the store at once. This is the case whenever constraint `p(`$\bar{x}$`)` has has the set semantic functional dependency $p(\bar{x}) :: \emptyset \rightsquigarrow \bar{x}$ for the current occurrence. An example is the constraint `gcd/1` from the `gcd` program in Example 14. Here the rule

```
gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

ensures one of the two `gcd/2` constraints (one must be active) will be deleted.[7] Therefore only one can ever actually be in the store at once, hence a `yesno` index structure may be used.

If constraint `p(`$\bar{x}$`)` has the set semantic functional dependency of the form $p(x_1, \ldots, x_i, x_{i+1}, \ldots, x_k) :: \{x_1, \ldots, x_i\} \rightsquigarrow \{x_1, ..., x_k\}$ for the current occurrence, then the compiler will generate a `tree` index structure, which in HAL is a balanced 234 tree. In this case the constraint `p(`$\bar{x}$`)` can be thought of as defining a function from the key $(x_1, \ldots, x_i)$ to the constraint `p(`$\bar{x}$`)` itself. For example, the `bounds(X,L,U)` constraint from the `bounds` program from Appendix A.1 has the functional dependency $bounds(X, L, U) :: \{X\} \rightsquigarrow \{X, L, U\}$ hence the compiler builds a 234 tree index structure with `X` as the key, and the numbered constraints `bounds(X,L,U) # Id` as the value.

Even without functional dependencies we can still use tree indexes. Instead of mapping a key to a unique constraint, a key maps to a list of constraints which share the same key. This is the purpose of the `tree*` index structure, which is a 234 tree from keys to list iterators. For example, the `X < Y` constraint from the `bounds` program has no functional dependencies between X and Y. For the lookup (`X < _`) we can use a `tree*` index, which maps the key X to a list iterator of all constraints of the form (`X < _`) in the store.

The big advantage of tree index structures is $O(\log(n))$ lookups compared with $O(n)$ lookups for unsorted lists used in the basic compilation. Another advantage of tree indexes is that it is possible to do a form of *tree index elimination*: suppose that for a constraint `p(X,Y,Z)` we need to generate indexes with keys (X, Y) and X corresponding to lookups `p(X,Y,_)` and `p(X,_,_)`. For the index where (X, Y) is the key, all keys of the form $(X, \_)$ will group together in the tree index because of HAL's default lexicographical term ordering. This eliminates the need for generating a separate index for the lookup `p(X,_,_)`, since we can use the index for `p(X,Y,_)` together with a special search, i.e. search for the largest subtree where the key for the root node is of the form $(X, \_)$ (this is an $O(\log(n))$ operation because we can prune large sections of the tree that are not of interest). From this subtree we build an iterator by collecting all nodes from the root of the subtree of the required form (we take advantage of the fact that all such nodes must be connected in the subtree).

Example 68 *For example, consider the following rules from the* `bounds` *program*

[6] These constraints do not have to be identical.

[7] Note that we assume that the active constraint is not present in the store. How/when this occurs is explained in Section 7.3.

```
X =< Y \ X =< Y <=> true.
X =< Y, bounds(X,LX,UX), bounds(Y,LY,UY) ==>
                    bounds(Y,LX,UY), bounds(X,LX,UY).
```

which require the lookups (`X =< Y`), (`X =< _`) and (`_ =< Y`). Usually the compiler will generate three indexes, one for each lookup. With our special kind of index elimination, we only require indexes for lookups (`X =< Y`) and (`_ =< Y`), and lookups of the form (`X =< _`) use the same index as the lookup for (`X =< Y`). □

The other main type of index structure is the hash table. Hash tables and the balanced tree indexes are interchangeable (i.e. a hash table index can be used instead of a tree index and vice versa). There are two main disadvantages of hash tables over tree index structures. Firstly, a hash table requires a hash function[8] over the domain of keys, and secondly, the tree index elimination is simply not applicable to hash tables, so usually more indexes are generated. The HAL CHR compiler uses a hash table if the types of each argument in the key have an associated hash function with it. This is achieved by a special typeclass `hash(T)` which defines a function of the same name. The type of the function is

```
:- func hash(T) = int.
```

The hash value of a tuple is just the exclusive-OR of the hash values of each element.

The advantage of hash tables is an average complexity of $O(1)$ for lookups, although a worse case of $O(n)$ is possible. This means that hash tables indexes on average have the same complexity as some of the best-case usages of attributed variable indexing, although hash tables generally have more overhead (e.g. computing the hash function etc.).

The final type of index structure is an unsorted list. A list is chosen for lookups of the form `p(_,...,_)`, i.e. when the key is empty. Such lookups require all `p(`$\bar{x}$`)` constraints currently in the store, so a list index is sensible.

Guarded index structures

Rather than generalising lookups, sometimes it makes sense to generate more specialised index structures. We briefly look at one such idea, namely *guarded indexes*.

Consider the lookup `p(X,X,_)`. Lookup generalisation will reduce this lookup to the more general lookup `p(X,Y,_)` along with the guard `X = Y` which is scheduled before the lookup. We could create a more specialised index structure by testing the guard `X = Y` *before* insertion into the (more specialised) index. In fact, this can be generalised to any type of guard, provided the variables in the guard appear in the lookup. For example, suppose join ordering schedules a

[8] Tree indexes similarly require a comparison function between terms. In HAL, virtually all types are comparable, so this is not a problem.

lookup `p(X,Y,_)` followed by the guard `X < Y`, then a similar specialised index is possible, where entries are only inserted into the index if the guard `X < Y` holds.

Guarded indexes are generally smaller, which makes lookups faster in the case of tree and list indexes. The cost of insertion is reduced, since not all constraints will be inserted into the index. Over use of guarded indexes may lead to a large number of index structures generated per constraint, which is undesirable. One solution is to use guarded indexes whenever lookup generalisation does not reduce the number of indexes. Currently the HAL CHR compiler does not support guarded index structures.

7.3 Global Optimisation

While local optimisation examines each occurrence in isolation, this section looks at global optimisation, which take into account the context of an occurrence (with respect to other occurrences) in order to make further improvements.

7.3.1 Continuation Optimisation

We can improve the simple strategy for joining the code generated for each occurrence of a constraint by noticing correspondences between rule matchings for various occurrences. If the current occurrence fails finds a match, we can use this information to show that subsequent occurrences will also fail to find a match. We define *fail continuation* as the next occurrence to be called when the current occurrence fails to match. Similarly if the occurrence successfully finds a match, we can also use this information. We define *success continuation* as the next occurrence to be called when the current occurrence finds at least one match. Under the basic compilation, both the success and fail continuations were simply the next occurrence.

Suppose we have two consecutive occurrences which we represent in the program normalised form (see Section 4.2.3).

$$\texttt{occ(p(}\bar{x}\texttt{),\_,}n\quad\texttt{,}H_1\texttt{,}H_2\texttt{,}g\texttt{,\_,\_)}$$
$$\texttt{occ(p(}\bar{y}\texttt{),\_,}n+1\texttt{,}H_1'\texttt{,}H_2'\texttt{,}g'\texttt{,\_,\_)}$$

Here n is the occurrence number, H_1 and H_2 are the non-deleted and deleted parts of the head (excluding the active constraint) and g is the guard. Similarly, H_1', H_2' and g' are heads and guard for the second occurrence. Let $c = H_1 \uplus H_2$ be the partners for occurrence n, and $c' = H_1' \uplus H_2'$ be the partners for occurrence $n+1$. We will assume that expressions g and g' already contain explicit existential quantification for variables not appearing in the rule head. We will also assume that the built-in solver used by the program is complete.

Suppose the CHR compiler can prove that there exists a renaming ρ such that $\bar{x} = \rho.\bar{y}$ and the following conditions hold

1. $c \subseteq \rho.c'$ (multiset inclusion)

2. $\mathcal{D} \models \rho.g' \rightarrow g$

Then, anytime the first occurrence fails to match the second occurrence will also fail to match, since the store has not changed meanwhile. Hence, the fail continuation for the first occurrence can skip over the second occurrence. We can use whatever reasoning we please to prove the implication. Currently, the HAL CHR compiler uses very simple implication reasoning about Herbrand equations (since the Herbrand solver is known to be complete) and identical CHR constraints.

Example 69 *Consider the (normalised) rule from the* `bounds` *program (see Appendix A.1) which contains the* 4^{th} *and* 5^{th} *occurrence of* `bounds`/3.

```
bounds(X,L1,U1)4, bounds(Y,L2,U2)5 <=> X = Y |
                                  bounds(X,max(L1,L2),min(U1,U2)).
```

Consider the substitution $\rho = (X = Y \wedge L1 = L2 \wedge U1 = U2)$ *which satisfies*

1. $\{bounds(Y, L2, U2)\} \subseteq \rho.\{bounds(X, L1, U1)\}$*; and*

2. $\mathcal{D} \models \rho.(X = Y) \rightarrow (X = Y)$.

Note that $\rho.(X = Y) \equiv (Y = X)$*. Hence, the* 5^{th} *occurrence will never succeed if the* 4^{th} *fails. Since if the* 4^{th} *succeeds the active constraint is deleted, the* 5^{th} *occurrence can be omitted entirely.* □

We can similarly improve success continuations. Suppose we have two consecutive occurrences, as before, but now the first occurrence does not delete the active constraint (otherwise the success continuation is irrelevant, since the active constraint would have been deleted). Let g_a be the maximal subset of g such that $vars(g_a) \subseteq \bar{x}$. Similarly we define g'_a as the maximal subset of g' such that $vars(g'_a) \subseteq \bar{y}$. If the compiler can prove that

$$\mathcal{D} \models (\bar{x} = \bar{y} \wedge g_a) \rightarrow \neg g'_a$$

then if the $p(\bar{x})$ occurrence succeeds the $p(\bar{y})$ occurrence will not. Hence, the success continuation of $p(\bar{x})$ can skip the $p(\bar{y})$ occurrence. Again, we can use whatever form of reasoning we please to prove the unsatisfiability.

Example 70 *Consider the two occurrences of* `p/2` *in the rules:*

```
p(X,Y), q(Y,Y,X,T) ==> X >= Y | ...
r(A,B,C), p(C,D) ==> C < D | ...
```

We see that

$$\mathcal{D} \models (X = C \wedge Y = D \wedge X \geq Y) \rightarrow \neg(C < D)$$

is clearly satisfiable and the success continuation of the first occurrence of `p/2` *can skip the second.* □

The conditions for success continuation optimisation are far stronger than fail continuation, and for this reason success continuation is less useful in practice.

There are subtle complications to consider for both kinds of continuation optimisation. Firstly, if a rule is prevented from firing because of the propagation history, then the success continuation (not the fail continuation) must be called. Secondly, for universal searches using iteration, there are difficulties stemming from deciding if the head of the rule fires or not, which is information that this optimisation relies upon. For the existential case there is no problem, since matching is already a mere `semidet` test. However a universal search may succeed multiple times, so some additional mechanism for recording the number of times a rule fires must be introduced. One possible solution is thread a counter through the code for the universal search, and count the number of times the search succeeds. If the counter is zero after the universal search code exists, then proceed with the fail continuation, otherwise proceed with the success continuation. Currently the HAL CHR compiler only implements continuation optimisation over existential searches.

Continuation optimisation was originally implemented for the SICStus CHR before the HAL CHR compiler, however (to the best of our knowledge) it was not formally reported at the time. All of the modern CHR compilers currently support some form of continuation optimisation.

7.3.2 Lateness Optimisations

Under the basic compilation, the top-level predicate for constraint `p`/k will perform three operations:

1. call `new`/1 to allocate a new identifier;
2. call `p_insert`/2 to insert the active constraint into the store; and
3. call `delay`/3 to set up appropriate delayed goals.

The aim of lateness optimisation is to postpone each of these operations until somewhere in the occurrence predicates if possible. The advantage is that if the active constraint is deleted before some operation occurs, then we save the cost of doing that operation. For example, if an active constraint is deleted before it is inserted into the store, then we save the cost of the insertion. In this section we look at the three kinds of lateness optimisations: *late ID*, *late storage* and *late delay*, which correspond to postponement of `new`/1, `p_insert`/2 and `delay`/3 respectively.

The late ID optimisation takes advantage of the fact that not all occurrences use the constraint identifier of the active constraint. We can therefore postpone the call to `new`/1 until the beginning of the first occurrence predicate that actually requires the identifier. The late ID optimisation works particularly well in conjunction with some of the overhead removal optimisations (see Section 7.2.1), which remove history checks, liveness checks, etc., requiring the active constraint's

identifier. Both of the calls to `p_insert` and `delay`/3 also require the identifier, so the late ID optimisation relies on *late storage* and *late delay* to be effective.

The late storage optimisation (unsurprisingly) relies on the late storage analysis of Chapter 5. Basically, the body of a rule (for a given occurrence) can observe the active constraint, then the active constraint must be inserted before the body is called. The question remains of where to place the call to `p_insert`. A naive approach is to place the call at the beginning the of first occurrence that requires the constraint to be stored, but this is problematic. Some index structures, e.g. the `yesno` index structure, require constraints to be inserted after (partner) deletion if applicable, otherwise two constraints may appear in the store at once.

A better approach is to call `p_insert` just before the body is called in the call-body predicate. The benefit is that if no matches are found, then the constraint will not be stored until even later occurrences (remember the later the better). However this does mean that `p_insert` may be called multiple times, e.g. once for each matching, as opposed to being called once under the basic compilation. We need a way of making sure the active constraint is inserted into the indexes exactly once. Similarly, it is possible that `p_delete` may be called when the active constraint has not actually been inserted. We need a way of only deleting the active constraint from the indexes if we know the active constraint has actually been inserted. The HAL CHR runtime system solves these problems by its implementation of the constraint identifier, which supports the following operations:

1. `store(Id)` – marks `Id` as belonging to a constraint that has been inserted (i.e. stored); and

2. `stored(Id)` – tests if `Id` belongs to a constraint that has already been inserted.

The pseudo code for `p_insert`[9] and `p_delete` is shown in Figure 7.7, which shows the usage of `store`/1 and `stored`/1.

Example 71 *Consider the compilation of the* `gcd` *program from Example 14. The first and second occurrences (*`gcd_1` *and* `gcd_2`*) delete the active constraint. Thus, the new* `gcd`/1 *constraint need not be stored before they are executed. It is only required to be stored during the code for the third occurrence (*`gcd_3`*). The calls to* `gcd_delete` *in* `gcd_1` *and* `gcd_3` *can be removed (since the compiler knows the active constraint has not been stored yet). Note that we must be careful to call* `gcd_insert` *after* `gcd_delete` *in occurrence three, in order to maintain the invariant that only one* `gcd`/1 *constraint exists in the store at once (so a* `yesno` *index may be used). Similarly, we have delayed the allocation of a new identifier*

[9]The interface for `p_insert` has changed slightly from the basic compilation. The old version, `p_insert`/2 expects two arguments, where the first points to C, which is some representation of the active constraint. The new version passes the arguments to the active constraint directly, avoiding the need to construct C each time `p_insert` is called.

```
p_insert(X1,...,Xk,Id) :-
    ( stored(Id) ->                 %% already stored?
        true                        %% no action
    ;   store(Id)                   %% else mark as stored
        <insert-into-indexes>       %% do insertion
    ).

p_delete(X1,...,Xk,Id) :-
    ( stored(Id) ->                 %% actually stored?
        <delete-from-indexes>       %% do deletion
    ;   true                        %% else no action
    ),
    kill(Id).                       %% mark as deleted
```

Figure 7.7: Pseudo code for `p_insert` and `p_delete`

for the active constraint until the start of the third occurrence. The code for `gcd`/1 *has now been simplified considerably, as illustrated in Figure 7.8, and is now more comparable to the hand-implemented version in Example 30. Note that we have inlined all join-loop and body-call predicates.* □

A simple form of the late storage optimisation was also implemented for the SICStus CHR compiler, where storage is delayed until the first occurrence that does not delete the active constraint. Our analysis-based approach is more powerful, as it often delays storage even further.

The final kind of lateness optimisation is *late delay*. We can postpone the call to `delay`/3 until the first occurrence where the body of the rule can "change" (i.e. further constrain) a variable appearing in the active constraint. Unfortunately, this is a fairly strong condition, and difficult to analyse accurately for. The reason is because CHRs manipulate global data (i.e. the CHR store), and therefore it is always possible to indirectly update variables mentioned anywhere in the body. For example, say the active constraint contains a variable X and the body of the current occurrence does not mention X, but the body does call some CHR constraint `q`/3. It is possible that when `q`/3 is active, it looks up another CHR constraint which contains variable X, and subsequently modifies it. This shows that calling any CHR constraint potentially modifies a variable. Currently, the late delay optimisation in HAL is very simple: we can postpone calling `delay`/3 if the body does not mention any variables (i.e. fully grounded) and does not call any CHR constraint.

Example 72 *Consider the first two rules from the* `leq` *program (see Example 1).*

```
leq(X,X)1              <=> true.
leq(X,Y)3 \ leq(X,Y)2 <=> true.
```

```
gcd(N) :-
    gcd_1(N).
gcd_1(N) :-
    (N = 0 ->                          %% Guard
        true,                          %% Body
        true                           %% success continuation
    ;   gcd_2(N,CN1)                   %% fail continuation
    ).
gcd_2(M) :-
    ( gcd_yesno_get_iterator(I),       %% get yesno iterator
      I = gcd(N) # Id,                 %% match?
      M >= N ->                        %% Guard
        gcd(M-N),                      %% Body
        true                           %% success continuation
    ;   gcd_3(M)                       %% fail continuation
    ).
gcd_3(N) :-
    new(Id),                           %% late ID
    ( gcd_yesno_get_iterator(I),       %% get yesno iterator
      I = gcd(M) # Id1,                %% match?
      M >= N ->                        %% Guard
        gcd_delete(Id1),               %% delete match
        gcd_insert(N,Id),              %% late storage
        gcd(M-N)                       %% Body
    ;   gcd_insert(N,Id),              %% late storage
    ).
```

Figure 7.8: Simplified code for `gcd`/1 with late storage and late ID optimisations

⋮

Both of these rules have ***true*** *as the body. Since the body does not mention any variable nor call any CHR constraints it is safe to postpone the call to* ***delay****/3 until the fourth occurrence.* □

7.3.3 Never Stored

A CHR constraint is *never-stored* (for a given occurrence) if it cannot appear in the CHR store. Information about never-stored is a consequence of the functional dependency analysis in Section 5.6, where the counts on the constraints are 0. For a given occurrence, if one of the matching partners is never-stored, then we know at compile time the search for partners will always fail. Hence an active constraint can skip such occurrences.

Example 73 *Consider the constraint* `add_color`/1 *from the ray tracer (see Example 59).*

```
add1 @ add_color(C1), color(C2) <=> C3 = C1 + C2, color(C3).
add2 @ add_color(C) <=> color(C).
```

Consider the occurrence for `color`/1 *in the first rule. Functional dependency analysis reveals that the* `add_color`/1 *constraint is never-stored (because of the* `add2` *rule). An active* `color`/1 *constraint will never match rule* `add1`*, since there can never be a matching* `add_color`/1 *in the store. Thus, we can remove the occurrence of* `color`/1 *from consideration when compiling the ray tracer program.* □

7.4 Experimental Results

In this section we show the benefit of CHR optimisation on several example programs and benchmarks. The current HAL CHR compiler supports most of the optimisations mention in this chapter, including

- overhead removal;
- join ordering and early guard scheduling;
- index selection;
- continuation optimisation;
- lateness optimisation; and
- never-stored optimisation

In order to implement these optimisations, the HAL CHR compiler uses information from HAL's type, mode and determinism declarations, and from the results of program analysis described in Chapter 5.

The HAL CHR compiler implements an ad hoc version of the late storage and set semantic functional dependency analyses described in Chapter 5.[10] The ad hoc versions of the analyses are essentially equivalent to the formal versions, e.g. functional dependency analysis still relies on detecting rules of a certain form and collecting constraints on the "shape" of the CHR store.

For the experiments, we will use the following test suite:

- `gcd` – The `gcd` program from Example 14, where the benchmark `gcd(`a`,`b`)` computes `gcd(`a`)`,`gcd(`b`)`.

[10]The HAL CHR compiler was fully implemented before work on the abstract interpretation for CHRs begun. The abstract interpretation framework and the analyses have been implemented as part of the K.U.Leuven CHR system (see [73] for more details) and will be included in any future CHR compiler for HAL/Mercury.

- `cycle` – Detects 5-cycles in directed graphs using the propagation rule

  ```
  edge(X,Y),edge(Y,Z),edge(Z,W),edge(W,T),edge(T,X) ==> cycle.
  ```

  The benchmark `cycle(`n`)` counts the number of cycles in a fully connected graph of n nodes. This example also appears in [47].

- `database` – An extended version of the `database` program in Example 15. Benchmark `database(`n`,`m`)` inserts and deletes n entries, and performs m lookups.

- `boolean` – A simple Boolean solver. Benchmark `queens(`n`)` finds a solution for the classic n-queens problem.

- `bounds` – The simple bounds propagation solver from Appendix A.1. Benchmark `job(`n`)` schedules jobs for bridge construction using data from [85]. Here, `job(113)` has solutions, and `job(79)` has none.

- `stack` – A stack data structure implemented purely in CHRs. Benchmark `stack(`n`)` pushes n elements, then pops the same n elements.

- `queue` – Similar to `stack` but implements a queue data structure. Benchmark `queue(`n`)` adds n elements, then retrieves the same n elements.

- `union` – An implementation of the naive *union find algorithm* in CHRs [72]. Benchmark `union(`n`)` creates, unions, and then finds n nodes.

- `union_opt` – An optimised implementation of union find with path compression and union-by-size [72].

- `graph` – A visual parser for directed graphs. Benchmark `graph(`n`)` constructs a fully connected graph with n nodes from geometric primitives such as circles and lines.

- `ray` – The `ray` program from Appendix A.2. Benchmark `ray` renders a 512×512 image of a scene consisting of 3 light sources, 7 spheres and 1 plane.

- `compiler` – Part of a bootstrapping CHR compiler[11] which analyses for never-stored and set-semantics. Benchmark `database(`n`)` analyses the `database` program, where each rule is (redundantly) repeated n times.

[11] This is a different bootstrapping compiler from Christian Holzbaur's version. The Holzbaur compiler has not been (fully) ported to HAL (to do so would require significant work), and hence it is not currently possible to use it as an example program.

Table 7.1: Statistics from each of the example programs

Prog.	$\|c\|$	<=>	\	==>	$\|r\|$	$\|H\|$
gcd	1	1	0	1	2	2
cycle	3	2	0	1	3	5
database	4	3	3	0	6	2
boolean	11	3	24	0	27	3
bounds	12	8	5	5	18	4
stack	4	5	1	0	6	3
queue	5	5	2	0	7	3
union	6	6	4	0	10	3
union_opt	6	8	3	0	11	3
graph	7	1	4	0	5	5
ray	16	13	13	4	30	3
compiler	30	9	42	16	67	5

Note that none of these programs use a built-in solver. Experiments on programs that do extend a built-in solver will be covered later in Section 8.6.

Table 7.1 summarises relevant information about each program. The $|c|$ column is the number of CHR constraints defined by each program. Next the <=> column is the number of simplification rules, \ is the number of simpagation rules, and ==> is the number of propagation rules. Next the $|r|$ column is the total number of rules. Finally, the $|H|$ column is the maximum head size (i.e. number of CHR constraints in the rule head) for any rule in the program. This is relevant for join ordering and early guard scheduling.

The results are shown in Table 7.2–Table 7.6 respectively. All timings are the average over 10 runs on a 1.2GHz AMD Athlon Processor with 1.5Gb of RAM running under Linux (Debian) with kernel version 2.4.22 and are given in milliseconds. SICStus Prolog 3.8.6 is run under compact code (no fastcode for Linux). We compare to SICStus CHRs where possible just to illustrate that the HAL CHR implementation is mature and competitive.

Table 7.2 shows timings for progressively more-optimised versions of the gcd program. The *basic* version is closest to the output of the basic compilation for gcd given in Example 36. The main difference is that the call to delay/3 has been removed.[12] In the version *+late* both late storage and late ID have been applied. This means that an active gcd/1 constraint will not be stored, nor will the constraint identifier be allocated, until the third occurrence. Since (for these benchmarks) the active constraints are generally deleted by the second occurrence, the lateness optimisation is highly beneficial, with an 80% improvement. The *+yesno* version uses a yesno index instead of a list index by taking advantage of the functional dependency $\texttt{gcd}(X) :: \emptyset \rightsquigarrow \{X\}$. This version is similar to the

[12]For technical reasons: since delay/3 is not defined for type int.

Table 7.2: Execution times (ms) for various optimised versions of the `gcd` program

Benchmark	*basic*	*+late*	*+yesno*	*hand*
`gcd(5000000,3)`	1566	321	65	41
`gcd(5000000,1)`	4739	962	196	126
`gcd(10000000,7)`	1348	274	56	35
`gcd(10000000,3)`	3148	642	131	85
geom. mean	2369	20%	4%	3%

compiled version of `gcd` shown in Example 71. The usage of a `yesno` over a `list` index is also beneficial, with a further 16% improvement. Finally, the *hand* versions is the human implemented `gcd` program of Example 30. Interestingly, the fully optimised *+yesno* version has comparable performance to the *hand* version. All other experiments in this section include overhead removal optimisations.[13]

Table 7.3 shows the benefit of using more efficient index structures to represent the CHR store. For this experiment we assume that all non-index related CHR optimisations (e.g join ordering, lateness, etc.) are enabled. The *list* version uses a `list` index for each individual CHR constraint. The *+tree* version uses `tree` indexes (and `yesno` indexes where appropriate). Finally, the `hash` version replaces the tree indexes with hash tables. Both `tree` and `hash` indexes are more efficient than `list` indexes in general, with a 93% improvement for trees and a 95% improvement for hash tables.

The `stack` and `ray` programs had the least benefit from `tree`/`hash` indexes. In fact, the `stack` program performs best with a `list` index. The reason is because of the way the `list` index is used by the `stack` program, i.e. the order elements are inserted into the list index (i.e. when an element is *pushed* onto the stack) is also the order elements are removed from the list. Therefore all `list` index operations are $O(1)$. On the other hand, the `queue` program retrieves elements from the tail of the `list` index, which results in much poorer performance. The `ray` program also showed little benefit from `tree`/`hash` indexing. The reason is that the CHR store never grows beyond a few dozen constraints, hence the benefit from using more efficient index structures is minimal.

Finally, the *SICS* version is provided to show how the HAL CHR compiler compares with an existing CHR implementation. All of the `bounds`, `graph` and `cycle` programs take too long to complete, so these are excluded from the results. For the programs that are included, the *SICS* version is considerably slower than the equivalent HAL *list* version. Note that this is partly because of the differences between the HAL compiler and SICStus Prolog, i.e. HAL is a faster programming language in general.

[13] All overhead removal optimisations are tightly integrated with the HAL CHR compiler. Thus disabling overhead removal is not currently possible.

Table 7.3: Execution times (ms) for various benchmarks testing indexing

Prog.	Benchmark	*list*	*+tree*	*+hash*	*SICS*
`bounds`	`queens(15)`	62330	914	570	–
`bounds`	`job(113)`	1530	94	61	–
`bounds`	`job(79)`	9555	526	339	–
`graph`	`graph(25)`	1746	295	300	–
`boolean`	`queens(8)`	9287	1419	886	5986
`boolean`	`queens(10)`	35543	3303	1680	13566
`cycle`	`cycle(14)`	4573	591	512	–
`database`	`database(5000,5)`	6816	56	29	102958
`database`	`database(10000,5)`	43558	123	61	434691
`stack`	`stack(100000)`	316	911	510	7610
`queue`	`queue(5000)`	4402	38	24	17003
`union`	`union(100)`	1181	208	137	10617
`union_opt`	`union(180)`	1137	172	89	12386
`ray`	`ray`	14087	13126	13262	436326
`compiler`	`database(100)`	6404	190	94	84893
	geom. mean	5478	7%	5%	652%*

Table 7.4 shows the benefit of join ordering and early guard scheduling. We test four programs with large rule heads: `cycle` (5 heads), `bounds` (4 heads), `compiler` (5 heads) and `graph` (5 heads). The other programs do not benefit from this optimisation, since either the maximum head size for any rule is less than 3, or coincidently, a good join order is always chosen. All other optimisations are enabled, except index selection (since join ordering affects index selection), so `list` indexes are used for a fairer comparison. The *orig* version does not apply join ordering or early guard scheduling (including guards that result from head normalisation). The order used in the joins is essentially arbitrary (i.e. what ever the compiler happens to chose). Not surprisingly, the resulting executables have very bad runtime performance. Some benchmarks, e.g. `cycle(12)`, were aborted after 10 minutes of execution time. The *+guard* version allows guards to be scheduled early, but still using the arbitrary join order. The benefit largely depends on the number of guards scheduled early. For example, we see a large improvement for the `cycle` program and a much smaller improvement for the `bounds` program. Finally the *+join* uses the brute-force algorithm to find a better join ordering according to the heuristic. Note that the *+join* version is equivalent to the *list* version in Table 7.3. The `cycle`, `bounds` and `graph` programs benefit, however join ordering chooses a worse order for the `compiler` program. This can be attributed to the join ordering heuristic finding a non-optimal join order based on (inaccurate) assumptions about selectivity or cardinality of constraints. In general, join ordering and early guard scheduling is highly beneficial, with a

Table 7.4: Execution times (ms) for various benchmarks testing join ordering and early guard scheduling

Prog.	Benchmark	*orig*	*+guard*	*+join*
`cycle`	`cycle(7)`	8673	32	16
`cycle`	`cycle(8)`	41824	98	51
`cycle`	`cycle(12)`	–	2519	1391
`bounds`	`queens(3)`	2458	2284	5
`bounds`	`queens(4)`	42108	40474	22
`compiler`	`database(40)`	8406	79	425
`compiler`	`database(60)`	28490	180	1427
`graph`	`graph(8)`	36279	20	15
`graph`	`graph(9)`	98103	37	26
`graph`	`graph(20)`	–	1796	688
	geom. mean*	20565	1%	0.2%

99.8% overall improvement.

Table 7.5 shows the benefit of continuation optimisation. All other optimisations are enabled, and all programs use hash indexes. The $-cont$ ($+cont$) has continuation optimisation disabled (enabled). The $+cont$ is equivalent to the $+hash$ version in Table 7.3. Overall, continuation optimisation sees a modest 3% improvement. This is mainly because only fail continuation was applicable to rules of the form

`p(`$\bar{x}$`) \ p(`$\bar{y}$`) <=> ...`

In this case, the second occurrence, $p(\bar{x})$, can be removed from consideration, thus removing the need for a (redundant) hash table lookup. The `union_opt` program had the most benefit from continuation optimisation, with a 10% improvement. This shows that continuation optimisation is worthwhile for some programs.

Table 7.6 shows the benefit of lateness optimisation. All other optimisations are enabled, and all programs use hash indexes. The $-late$ ($+late$) has lateness optimisation disabled (enabled).[14] The $+late$ is equivalent to the $+late$ version in Table 7.3. We see an overall 50% improvement, showing that lateness optimisation is worthwhile. Note that some of the improvement is because of indirect consequences of disabling late storage. For example, without late storage, the interpretation of functional dependencies and never-stored changes, thus weaker indexes are be used and never-stored optimisation is effectively disabled. This explains why some programs, e.g. the `boolean` program, benefit greatly from lateness optimisation.

[14]Note that this optimisation cannot be disabled for CHR constraints with at least one argument with an '`out`' mode. This is because such constraints must be never-stored.

Table 7.5: Execution times (ms) for various benchmarks testing continuation optimisation

Prog.	Benchmark	$-cont$	$+cont$
`bounds`	`queens(18)`	14591	14337
`boolean`	`queens(12)`	8588	8482
`database`	`database(100000,1)`	428	405
`database`	`database(100000,5)`	641	625
`stack`	`stack(500000)`	2581	2513
`queue`	`queue(500000)`	2509	2482
`union`	`union(160)`	560	555
`union_opt`	`union(500)`	768	695
`ray`	`ray`	13324	13262
`compiler`	`database(300)`	1223	1210
	geom. mean	2086	97%

Table 7.6: Execution times (ms) for various benchmarks testing lateness optimisation

Prog.	Benchmark	$-late$	$+late$
`bounds`	`queens(18)`	27144	14337
`graph`	`graph(30)`	603	589
`boolean`	`queens(12)`	176337	8482
`cycle`	`cycle(14)`	526	512
`database`	`database(100000,5)`	1660	625
`stack`	`stack(500000)`	2967	2513
`queue`	`queue(500000)`	2950	2482
`union`	`union(160)`	578	555
`union_opt`	`union(500)`	947	695
`ray`	`ray`	29799	13262
`compiler`	`database(300)`	5078	1210
	geom. mean	3709	50%

7.5 Summary

In this chapter we have optimised the basic compilation of Chapter 4 which has greatly improved the runtime performance of many CHR programs. Several optimisations were discussed, and these could be grouped into two categories: local optimisation, which aims to optimise each occurrence in isolation, and global optimisation, which applies further improvements based on the context of an occurrence with respect to other occurrences.

Local optimisation includes overhead removal, join ordering and index selection. Overhead removal attempts to specialise universal searches to existential searches when only one match is required. Also, propagation histories can be removed altogether under certain conditions, and other forms of overhead, e.g. liveness checks, difference checks, etc., can sometimes be removed with help from information available to the compiler.

While overhead removal is useful, the core of compiling CHRs is a multi-way join compilation. But, unlike the usual database case, we have no information on the cardinality of relations and index selectivity. We show how to use type and mode information to compile efficient joins, and automatically utilise appropriate indexes for supporting the joins. We show how set semantics, functional dependencies and symmetries can improve this compilation process. The HAL CHR compiler which applies these techniques produces highly efficient CHR executables.

Global optimisation enriches compilation further with continuation optimisation, lateness optimisation and never stored optimisation. Continuation optimisation attempts to prove that a failed/successful match implies later occurrences will fail to match, and hence can be avoided. Lateness optimisation attempts to postpone active constraint initialisation as far as possible, with the hope that the active constraint will be deleted before the initialisation occurs. Finally the never-stored optimisation further removes occurrences from consideration by proving certain CHR constraints will never be present in the CHR store.

Chapter 8

Extending Solvers

8.1 Introduction

A CHR program *extends* another solver if the CHR constraints are allowed to contain non-fixed solver variables at runtime. Typically, the CHR constraints are viewed as new constraints defined in terms of existing constraints, hence we have "extended" the existing solver by adding new constraints. For example, the `leq` solver from Example 1 *extends* a Herbrand solver, because the `leq`/2 constraint is defined in terms of Herbrand constraints, and `leq`/2 constraints are allowed to be non-fixed at runtime.

For a CHR program that extends another solver, the **Solve** transition may wakeup a non-empty subset of the CHR store. So far we have only briefly covered the implementation of CHR programs that extend other solvers (see Section 4.3.3). In this chapter we explore the implementation of CHRs extending other solvers in far greater detail, including:

- formalising the *wakeup policy* (see Definition 10) used by most CHR implementations (including Prolog implementations);
- Optimising the `delay`/3 code by various kinds of specialisation;
- Building indexes over non-fixed data;
- Implementing dynamic scheduling in HAL.

We also provide some experimental results that show the effectiveness of various approaches.

Under the basic compilation, we relied upon the existence of a simple form of dynamic scheduling to wakeup an appropriate subset of the CHR store during a **Solve** transition. After a call to the special predicate `delay(`$Term$`,`Id`,`$Goal$`)` the *Goal* (which is the first occurrence predicate) is called each time a variable in $Term$ (which is the CHR constraint) "changes". In this chapter we formally define what "changes" means, and show that our implementation satisfies the necessary conditions for a wakeup policy.

One of the aims of this thesis is to optimise the compilation of CHRs. Often, the set of CHR constraints woken up using our simple form of dynamic scheduling is far greater that the minimum set that is actually required. This is clearly inefficient, since it means redundant rule checking. We will discuss ways of reducing the size of the set of constraints that are woken up based on a simple analysis of the guards, and some additional help from the built-in solver.

Another one of our aims is build a CHR system that can extend arbitrary solvers. While the semantic basis of CHRs assumes an arbitrary built-in constraint solver, in practice, implementations of CHRs only extend Herbrand equation solvers. In this chapter we look at generalising the solver interface, so CHRs can extend any solver that implements that interface. We will also look at some examples implementations of the interface.

One issue that has been delayed until this chapter is building index structures over solver variables. Efficient index structures are an essential part of any optimised CHR implementation. However, indexes implicitly ask equality constraints, and the result of an ask equality can change throughout a derivation if the data is non-fixed. In this chapter we discuss a way of fixing index structures so that they do not become corrupted when the data changes. Once again our approach is based on dynamic scheduling.

The rest of the chapter is organised as follows. Section 8.2 formalises and then proves the correctness of the *wakeup policy* based on the simple dynamic scheduling. Section 8.3 discusses how to use specialisation to reduce the number of constraints that are woken up during a **Solve** transition. Section 8.4 discusses the use of indexes for CHRs that use solver variables. Section 8.5 discusses how the `delay` interface is implemented by solvers in HAL. Section 8.6 presents the results of some experiments illustrating the benefits of specialised rechecking and indexes. Finally, we conclude.

8.2 Wakeup Policy

In this section we officially formalise the usage of `delay`/3 as described in Section 4.3.3 as a *wakeup policy* as defined by Definition 10. This involves showing that the set of constraints woken up by our implementation satisfies the *lower* and *upper* bound conditions of a wakeup policy. For simplicity, we will initially assume the complete built-in solvers (e.g. a Herbrand solver), thus ($\mathcal{D} \models_{\mathcal{S}}$) is equivalent to ($\mathcal{D} \models$). We will discuss incomplete solvers later in this section.

Consider the call `delay(`$Term$`,`Id`,`$Goal$`)`, first we need to formally define exactly when $Goal$ is called, i.e. when $Term$ has "changed". A $Term$ has changed if there exists a variable $v \in vars(Term)$ such that v has "changed". We define the set of changed variables $changed(c, B)$ as the following.

Definition 48 (Changed) *Let c be a built-in constraint, and let B be a built-in store. Let $\bar{x}$ be a set of variables satisfying*

$$\mathcal{D} \models \neg\forall_{\bar{x}}(\bar{\exists}_{\bar{x}} B \rightarrow \bar{\exists}_{\bar{x}}(B \wedge c)) \tag{8.1}$$

and all $\bar{y} \subset \bar{x}$ *satisfy*

$$\mathcal{D} \models \forall_{\bar{y}}(\bar{\exists}_{\bar{y}} B \rightarrow \bar{\exists}_{\bar{y}}(B \wedge c)) \tag{8.2}$$

then $\bar{x} \subseteq changed(c, B)$*. The set* $changed(c, B)$ *is constructed by considering all possible* $\bar{x}$*.* □

Informally, (8.1) ensures that for some set of variables $\bar{x}$, there exists values for $\bar{x}$ such that a solution to B exists, but a solution to $B \wedge c$ does not exist for the same values. Therefore, the values we can assign to $\bar{x}$ has *changed*. The second part of the definition ensures that $\bar{x}$ is a *minimal* set satisfying (8.1), i.e. there does not exists a proper subset of $\bar{x}$ that has also changed.

Example 74 *Consider an execution state with an empty built-in store* ($B = true$) *over a Boolean solver, and let* X*,* Y *and* Z *be (unconstrained) Boolean variables. Suppose we add the constraint* $X = Y$ *into the built-in store by a* ***Solve*** *transition. We see that*

$$\mathcal{D} \models \neg\forall X \forall Y(true \rightarrow \exists Z(X = Y))$$

clearly holds (try $X = 0$ *and* $Y = 1$*), therefore the set of variables* $\{X, Y\}$ *has changed. Also, each proper subset of* $\{X, Y\}$*, i.e.* $\emptyset$*,* $\{X\}$ *and* $\{Y\}$ *have not changed, e.g. for the subset* $\{X\}$ *we see that*

$$\mathcal{D} \models \forall X(true \rightarrow \exists Z \exists Y(X = Y))$$

holds (try $Y = X$ *for given* X*), etc. Therefore* $\{X, Y\} \subseteq changed(X = Y, true)$*.*

Proving that $Z \notin changed(X = Y, true)$ *is slightly more difficult. We can easily prove that* $\{Z\} \not\subseteq changed(X = Y, true)$ *by showing that*

$$\mathcal{D} \models \neg\forall Z(true \rightarrow \exists X \exists Y(X = Y))$$

does not hold (try $X = 0$ *and* $Y = 0$ *for given* Z*). Also, for any set of variables containing* Z *that has changed, we can find a subset not containing* Z *that has also changed. Therefore* $Z \notin changed(X = Y, true)$*, so* $changed(X = Y, true) = \{X, Y\}$*.* □

It is usually the case that $vars(c) \subseteq changed(c, B)$, unless c is already implied by B.

We can use Definition 48 to formally define the subset of the CHR store that is woken up by `delay`/3, i.e. the *wakeup policy* of the basic compilation.

Definition 49 *Let* σ_i *and* σ_{i+1} *be* ω_r *execution states such that* $\sigma_i \rightarrowtail_{solve} \sigma_{i+1}$*. Let* c *be the built-in constraint such that* $B_{i+1} = c \wedge B_i$*, then we define*

$$delay\_wakeups(S_i, c, B_i) = \{d \in S_i \mid vars(d) \cap changed(c, B_i) \neq \emptyset\}$$

□

In other words $delay\_wakeups(S_i, c, B_i)$ is the subset of S_i containing constraints which contains at least one variable that changed as a result of adding c into the built-in store.

We now give a lemma which shows that $delay\_wakeups(S, c, B)$ satisfies the conditions for a wakeup policy in Definition 10.

Lemma 14 *For all consecutive ω_r states σ_i and σ_{i+1} such that $\sigma_i \rightarrowtail_{solve} \sigma_{i+1}$, the set $delay\_wakeups(S, c, B_i)$ satisfies the conditions for a wakeup policy in Definition 10, otherwise $\sigma_{i+1} = false$.*

Proof. The set $delay\_wakeups(\sigma_i)$ must satisfy the *lower bound* and *upper bound* conditions from Definition 10.

- *lower bound*: Direct Proof. Suppose that there exists a $M = H_1 +\!\!+ H_2 \subseteq S_i$, a rule $(r \ @ \ H_1' \setminus H_2' \iff g \mid C)$ and a matching substitution θ such that

$$\begin{cases} cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ \mathcal{D} \models \neg(B_i \rightarrow \exists_r(\theta \wedge g)) \\ \mathcal{D} \models (B_i \wedge c \rightarrow \exists_r(\theta \wedge g)) \end{cases} \tag{8.3}$$

  Let ϑ be a partial solution to B_i on the non-existential variables in $(\theta \wedge g)$ such that ϑ is not a solution of the guard, i.e.

$$\mathcal{D} \models \neg\bar{\exists}_\vartheta \vartheta(\theta \wedge g)$$

  One such ϑ must exist, otherwise the guard g must be equivalent to the trivial constraint $true$ by definition. Clearly, such a g could never satisfy (8.3).

  Then by (8.3)

$$\mathcal{D} \models \neg\bar{\exists}_\vartheta \vartheta(B_i \wedge c)$$

  and

$$\mathcal{D} \models \bar{\exists}_\vartheta \vartheta(B_i)$$

  Hence

$$\mathcal{D} \models \neg\vartheta(\bar{\exists}_\vartheta B_i \rightarrow \bar{\exists}_\vartheta(B_i \wedge c))$$

  Therefore

$$\mathcal{D} \models \neg\bar{\forall}_\vartheta(\bar{\exists}_\vartheta B_i \rightarrow \bar{\exists}_\vartheta(B_i \wedge c))$$

  by existential introduction.

  This is in the required form (8.1) from Definition 48, where $\bar{x} = vars(\vartheta)$ is the set of all non-existential variables in $(\theta \wedge g)$.

  Clearly, there exists a subset $\bar{y}$ of $vars(\vartheta)$ such that $\bar{y} \subseteq changed(c, B_i)$. If $\bar{y} = \emptyset$, then

$$\mathcal{D} \models \neg(\bar{\exists}_\emptyset B_i \rightarrow \bar{\exists}_\emptyset(B_i \wedge c))$$

which means $(B_i \wedge c)$ is unsatisfiable, hence $\sigma_{i+1} = false$.

Otherwise $\bar{y} \neq \emptyset$. Since $\bar{y}$ are non-existential variables from the guard by construction, then $\bar{y} \subseteq vars(M)$. Thus, there exists a $h \in M$ such that $vars(h) \cap changed(c, B_i) \neq \emptyset$. Therefore $h \in delay\_wakeups(S, c, B_i)$, and the *lower bound* condition is satisfied.

- *upper bound*: By contradiction.

  We show that if $vars(h) \subseteq fixed(B_i)$, then $h \notin delay\_wakeups(S, c, B)$ because $vars(h) \cap changed(c, B_i) = \emptyset$.

  Assume that $vars(h) \cap changed(c, B_i) \neq \emptyset$, i.e. there is a variable x such that $x \in vars(h)$ (and therefore $x \in fixed(B_i)$) and $x \in changed(c, B_i)$. Therefore there exists a set of variables $\bar{x}$, with $x \in \bar{x}$, such that $\bar{x}$ satisfies (8.1) and $\bar{y} = \bar{x} - \{x\}$ satisfies (8.2) from Definition 48.

  From (8.1) we have that

$$\mathcal{D} \models \exists x \exists_{\bar{y}} \neg(\bar{\exists}_{\bar{x}} B \rightarrow \bar{\exists}_{\bar{x}}(B \wedge c)) \tag{8.4}$$

  Let $\vartheta_{\bar{x}} = \vartheta_x.\vartheta_{\bar{y}}$ be a partial solution to (8.4). Hence

$$\mathcal{D} \models \vartheta_x \vartheta_{\bar{y}} \neg(\bar{\exists}_{\emptyset} B \rightarrow \bar{\exists}_{\emptyset}(B \wedge c)) \tag{8.5}$$

  From (8.2) we have that

$$\mathcal{D} \models \forall_{\bar{y}}(\bar{\exists}_{\bar{y}} B \rightarrow \bar{\exists}_{\bar{y}}(B \wedge c))$$

  and therefore

$$\mathcal{D} \models \vartheta_{\bar{y}}(\bar{\exists}_{\emptyset} \vartheta_x B \rightarrow \bar{\exists}_{\emptyset} \vartheta'_x(B \wedge c)) \tag{8.6}$$

  using the same $\vartheta_{\bar{y}}$ and ϑ_x from above, and ϑ'_x a new partial solution for x.

  We note that $\vartheta_x \neq \vartheta'_x$, otherwise (8.6) contradicts (8.5).

  From (8.5) we derive

$$\mathcal{D} \models \vartheta_x \vartheta_{\bar{y}}(\bar{\exists}_{\emptyset} B) \tag{8.7}$$

  and from this and (8.6) we derive

$$\mathcal{D} \models \vartheta'_x \vartheta_{\bar{y}}(\bar{\exists}_{\emptyset}(B \wedge c))$$

  which derives

$$\mathcal{D} \models \vartheta'_x \vartheta_{\bar{y}}(\bar{\exists}_{\emptyset} B) \tag{8.8}$$

  We combine (8.7), (8.8) and $\vartheta_x \neq \vartheta'_x$ together with existential introduction to derive

$$\mathcal{D} \models \exists x \exists \rho(x)(\bar{\exists}_x B \wedge \bar{\exists}_{\rho(x)} \rho(B) \wedge x \neq \rho(x))$$

  for a renaming ρ. This clearly contradicts Definition 9 (for a complete solver), therefore $x \notin fixed(B)$ which contradicts our original assumption. Therefore the upper bound condition is satisfied.

Therefore *delay_wakeups* satisfies the conditions for a wakeup policy in Definition 10. □

In practice, the solver usually decides the set of changed variables based on those that have been internally updated by the addition of the new tell constraint c. For example, in a Herbrand solver using the WAM representation [87, 6], the set of changed variables are those whose pointer representation has been updated by a unification.

For an incomplete solver, the set of woken up constraints is usually smaller than that of *delay_wakeups* from Definition 49. For example, HAL's finite domain solver defines the set of *changed* variables to be those whose domains have changed (i.e. reduced). This is weaker definition to that from Definition 48. For example, consider a built-in $B \equiv (X = Y)$ and the new constraint $c \equiv (Y = Z)$. If the domains of X, Y and Z are all equal, then the addition of c into B will cause no new domain propagation, and thus none of X, Y and Z have "changed". Under the stronger Definition 48, all of X, Y and Z must have changed.

The task of showing that a wakeup policy from an incomplete solver is correct with respect to the definitions is solver dependent. However, most of the time we can use more ad hoc reasoning to argue for correctness. For example, the ask constraints provided by the finite domain solver are incomplete, since they only consider the domains of input arguments, e.g. ask version of $X = Y$ only succeeds if the domains of X and Y are the same singleton set. Therefore, a finite domain ask constraint in a guard can only succeed where it previously failed if the domain of (at least) one of variables change. Therefore waking up constraints on domain change is appropriate.

8.3 Rechecking Rules

In this section we look at ways of improving the simple strategy of rechecking all occurrences when a variable changes.

8.3.1 Optimising Delay

The wakeup policy *delay_wakeups* of Section 8.2 is often inefficient, since constraints are woken up on *any* change on *any* variable that appears in that constraint. A better approach is to wakeup constraints on more specialised conditions, based on analysis of the actual guards to be tested. Our aim is to reduce the number of constraints that are woken up when an actual change does occur, which is beneficial since this means less work (re)checking if rules can fire. The complexity arises from deciding if/when more specialised conditions can be used.

The simplest form of this optimisation occurs when all guards for a given CHR constraint are trivial, i.e., all guards are *true*.

Example 75 *Consider the following rules from Example 59 from the ray tracer.*

```
add1 @ add_color(C1), color(C2) <=> C3 = C1 + C2, color(C3).
add2 @ add_color(C) <=> color(C).
```

The guard for each of these rules is trivial, hence never fails, and is never affected to be addition of any tell constraints into the store. Assuming the `color`/1 *constraint was allowed to be non-fixed at runtime, then we never need to wakeup such a constraint since for any built-in constraint c both*

$$\mathcal{D} \not\models_{\mathcal{S}} (B \rightarrow true) \quad \text{and} \quad \mathcal{D} \models_{\mathcal{S}} (B \wedge c \rightarrow true)$$

can never hold.[1] *Therefore a call to* `delay`/3 *is never required.* □

In general this optimisation requires the ability to delay goals on more specialised conditions, so we need a more sophisticated version of dynamic scheduling. Fortunately, HAL's actual implementation of dynamic scheduling is far richer than the simple version of `delay`/3 we introduced in Section 4.3.3. In HAL, `delay`/3 allows goals to be delayed on more specific conditions other than "variable has changed". For example, it is possible to delay goals on conditions such as "the variable X has become ground" or "the lower bound of X has changed", etc. The syntax for the new version of dynamic scheduling is `delay(`$Event$`,`Id`,`$Goal$`)` where $Event$ is a *solver event* (as opposed to a solver term, as in the previous version of `delay`/3). Solver events are of the form `event(`$Term$`)` where $Term$ is a solver term, and `event` is the name of the event.

Example 76 *Consider HAL's finite domain solver which supports the new version of dynamic scheduling. The solver events supported by this solver (the usual ones for a finite domain solver) are*

`fixed(X)`	*the domain of* `X` *is reduced to a single value.*
`lbc(X)`	*the lower bound of* `X` *changes (increases).*
`ubc(X)`	*the upper bound of* `X` *changes (decreases).*
`dc(X)`	*the domain of* `X` *changes (reduces).*

Note that these solver events are not mutually exclusive. For example, if the domain of `X` *changes from* $\{1, 3, 5\}$ *to* $\{1\}$*, then the events* `fixed(X)`*,* `ubc(X)` *and* `dc(X)` *all hold.*

HAL's Herbrand solver supports two solver events, which are

`touched(X)`	*variable* `X` *has been unified (to a variable or non-variable).*
`bound(X)`	*variable* `X` *has been unified with a non-variable.*

□

[1] This can never hold for any built-in solver implemented in HAL, since the constraint *true* is treated specially. In general, it is possible to define an incomplete test ($\mathcal{D} \models_{\mathcal{S}}$) that violates this assumption.

HAL's finite domain and Herbrand solvers will be the focus for the rest of this section. The main reason is that these are the only solvers that have been fully implemented (with dynamic scheduling) so far.

We will also assume that some solver events are shared by multiple solvers. The solver event `changed(`X`)` will wakeup the goal each time a variable in solver term X changes. In fact, the call `delay(`$Term$`,`Id`,`$Goal$`)` under the old dynamic scheduling is equivalent to the call `delay(changed(`$Term$`),`Id`,`$Goal$`)` under the new version.

The naive re-execution of the basic compilation rechecks every possible occurrence each time the solver state for the variables involved changes. This often causes no new rules to fire. We can improve upon this by (a) determining a set of solver events which more accurately signal the possible entailment of an ask constraint, and (b) building code that only reconsiders occurrences associated to those ask constraints.

Example 77 *Consider the following implementation of the ask '$\leq$' constraint for the* ***fdint*** *finite domain solver:*

```
'ask_=<'(X,Y) :-
    UBX = fd_max(X),        %% get current X upper bound
    LBY = fd_min(Y),        %% get current Y lower bound
    UBX =< LBY.             %% integer comparison
```

where functions `fd_max`/1 *and* `fd_min`/1*, respectively, return the (integer) upper and lower bounds of a variable's current domain. Note that this is an incomplete test since, even if the constraint $X \leq Y$ has been added to the store, the ask constraint may not succeed. Given this definition, the answer to the ask constraint will only change if either of the solver events* ***ubc(X)*** *or* ***lbc(Y)*** *occur. Other events will never signal a possible change in the answer to the ask constraint, unless one of these events also occurs. Note that using* ***ubc(X)*** *and* ***lbc(Y)*** *is much more accurate than using* ***changed(X)*** *and* ***changed(Y)*** *as done under the basic compilation.*

Consider now the following implementation of the ask `=`/2 *constraint for the* ***fdint*** *solver:*

```
X == Y :-
    val(X,Value),           %% get fixed value for X
    val(Y,Value).           %% is same fixed value for Y?
```

The ask constraint for $X = Y$ (`X == Y`) will only succeed if X and Y are both fixed to the same value. As in most finite domain solvers the implementation of the ask `=`/2 *constraint is quite incomplete. Given this definition, the only solver events where we should recheck a $X = Y$ guard are* `fixed(X)` *or* `fixed(Y)`*.*

A list of finite domain ask constraints and their corresponding solver events is given in Figure (8.1). $X \geq Y$ ask constraints may change answers on `lbc(X)` *or* `ubc(Y)`*, the opposite for $\leq$ constraints. Disequality ask constraints ($\neq$) may change answers on any generic domain change.* □

Constraint	*Events*
$X = Y$	`fixed(X), fixed(Y)`
$X \neq Y$	`dc(X), dc(Y)`
$X \leq Y$	`ubc(X), lbc(Y)`
$X \geq Y$	`lbc(X), ubc(Y)`

Figure 8.1: Relationship between finite domain ask constraints and solver events.

In order to allow each solver to provide a list of the relevant solver events for each ask constraint, we extend the `asks` declaration (introduced in Section 4.5.1) as follows

`:-` <*ask-constraint*> `asks` <*tell-constraint*> [`wakes` <*solver-event*>$*$].

The first part of the declaration is the same as before, where a mapping between an ask and tell constraint is defined. The new part, prefixed by token '`wakes`', provides a list of solver events that may cause the ask constraint to succeed. We shall refer to this as the *wakes list*. The wakes list is optional, and by default it will be empty. The HAL compiler uses type analysis to ensure that each solver event is supported by the solver.

Example 78 *Our finite domain solver* ***fdint*** *provides the following declarations:*

```
:- 'ask_=<'(X,Y) asks X =< Y wakes [ubc(X),lbc(Y)].
:- X == Y asks X = Y wakes [fixed(X),fixed(Y)].
```

indicating that the ask ***X =< Y*** *constraint only needs to be rechecked whenever the* ***ubc(X)*** *or* ***lbc(Y)*** *events occur. Similarly, the ask* ***X = Y*** *constraint only needs to be rechecked whenever the* ***fixed(X)*** *or* ***fixed(Y)*** *events occur.* □

As before, explicit existential quantification is allowed in asks declarations.

Example 79 *The extended form of the* ***asks*** *declaration given in Example 44 is*

```
:- nonneg(Y) asks exists [Z] Z = 2^Y wakes [lbc(Y)].
```

Likewise, the extended form of ***asks*** *declarations for the* ***bound_f*** *functions (see Section 4.5.2) are*

```
:- bound_f(X) asks exists [Y1,...,Yn] X = f(Y1,...,Yn)
                                          wakes [bound(X)].
```

□

In creating a wakes list, the solver writer should endeavour to use a *complete* set of solver events, so that any change in the solver state is captured by a solver event which could indicate that the ask constraint now succeeds.

Example 80 *The following* `asks` *declaration has an incomplete wakes list.*

```
:- 'ask_=<'(X,Y) asks X =< Y wakes [fixed(X),lbc(Y)].
```

An upper bound change on `X` *that doesn't fix* `X` *may change the answer to the ask constraint, therefore this declaration is incomplete.* □

Failure to provide a complete list will result in a system that does not meet the lower bound condition of a wakeup policy (see Definition 10), hence is a violation of the refined semantics.[2] The completeness property is therefore important.

The solver writer should also endeavour to use a *minimal* set of solver events in order to generate more efficient code. When a wakes list is not minimal the resulting delayed goals may be called more often than required.

The extended `asks` declarations allow the HAL CHR compiler to determine more accurately which occurrences need to be rechecked for which solver events. In order to do this, the compiler examines every possible solver event for each variable, and determines the subset of occurrences that must be examined if a solver event become true. It then uses this information to produce more specialised code when possible. For example, suppose the compiler needs to generate code that sets up the delayed goals for a CHR constraint `p(`A`,`B`,`C`)`. Suppose that all of the guards that use variable A only need to be rechecked on a `lbc(`A`)` event, then we can replace the general call to `delay(changed(`A`),`Id`,p_1(`A`,`B`,`C`))` with the more specialised call `delay(lbc(`A`),`Id`,p_1(`A`,`B`,`C`))`. The advantage is that if only the upper bound of A changes, we avoid reactivating the constraint `p(`A`,`B`,`C`)` over again.

Example 81 *Consider a Boolean solver implemented in CHRs extending a Herbrand solver (over integers 1 or 0 representing* $true$ *and* $false$ *respectively). The definition of an* `and(`X`,`Y`,`Z`)` *constraint is below, where* Z *is the result of the logical-AND of* X *and* Y*.*

```
and(X,Y,Z) <=> X = 0         | Z = 0.
and(X,Y,Z) <=> Y = 0         | Z = 0.
and(X,Y,Z) <=> X = 1, Y = 1  | Z = 1.
and(X,Y,Z) <=> Z = 1         | X = 1, Y = 1.
and(X,Y,Z) <=> Z = 0         | not_both(X,Y).
```

The constraint `not_both(`X`,`Y`)` *(defined elsewhere) fails if both* X *and* Y *are true.*

All of the guards are conjunctions of constraints of the form $V = 1$ *or* $V = 0$*, where* V *represents either* X*,* Y *or* Z*. The wakes list for guards of that form is* `[bound(V)]`*, as in Example 79. The result is (more specialised) predicate* `and_delay` *which sets up the delayed goals for* `and`/3*.*

[2]Note that this is still correct with respect to the declarative semantics of CHRs, in the sense that the resulting final state is still logically equivalent to the original goal. The problem is that the CHR store might not be as reduced as it otherwise should be.

```
and_delay(X,Y,Z,Id) :-
    Goal = and_1(X,Y,Z,Id),
    delay(bound(X),Id,Goal),
    delay(bound(Y),Id,Goal),
    delay(bound(Z),Id,Goal).
```

A call to `and_delay(X,Y,Z,Id)` *replaces the call to* `delay`/3 *under the basic compilation.* □

Sometimes an argument is never mentioned in any guard. For example, the argument $C2$ for is not mentioned in any guard of the `color`/1 constraint in Example 75. This removes the need to call `delay`/3 altogether. Another example is the Fibonacci constraint of Example 16. Assuming that both arguments may be solver variables, there is no need to call `delay`/3 on the second argument F of a `fib`/2 constraint. This is because the second argument F is not mentioned in any guard.

8.3.2 Accurate Specialisation

If a CHR program does not rely on the refined operational semantics, e.g. it is confluent under the theoretical semantics, then we can specialise the implementation of `delay` even further. In particular, we can specialise the delayed goal so that it does not check occurrences the compiler can prove will not match, even after a variable has changed.

Example 82 *Consider the definition of the* `and`/3 *constraint in Example 81, and consider the constraint* `and(A,B,C)` *where* A, B *and* C *are unbound Herbrand variables.*

Adding the constraint $C = 0$ *into the built-in store causes the* `bound(C)` *solver event to occur. Under the implementation of* `and_delay` *presented in Example 81, the goal* `and_1(A,B,0,Id)` *is called, which checks all occurrences* 1–4 *before checking occurrence* 5 *(which finally fires the rule). This means a total of* 5 *occurrences were checked in total. Observe that the guards for occurrences* 1–3 *do not even mention the third argument* Z*, therefore the reactivated constraint need not check these occurrences. If we avoid checking these occurrences, then a total of* 2 *occurrences are checked in total.* □

It is generally desirable to avoid rechecking occurrences which we know are still doomed to fail, since this avoids redundant work. The optimised re-execution can be *arbitrarily* faster than the naive approach since checking an occurrence could be arbitrarily difficult.

We can generalise this optimisation as follows. For each solver event e the compiler determines a sequence of occurrences $n_1, ..., n_m$ that need to be rechecked when e occurs (based on the analysis of the guards). Next a specialised *wakeup predicate* is generated that checks only those occurrences. We use the name

p_wakeup_n_1_..._n_m to represent a specialised wakeup goals that only checks occurrences $n_1, ..., n_m$.

There are however tradeoffs in creating the specialised delay code. In order to avoid code explosion, the compiler separates each occurrence predicate as opposed to chaining them together. This means that occurrence predicate p_n no longer calls occurrence predicate p_$(n+1)$ (as is the case under the basic compilation). The control logic for deciding if the next occurrence should be called is moved to the caller of the occurrence predicates, i.e. the wakeup predicate(s) and the top-level predicate (which now calls all occurrences). We can straightforwardly create the different sequences of occurrences that are required for each solver event.

Example 83 *Consider the following* ***min*** *program similar to that in Example 37 except with a third rule for adding redundant constraints and a fourth rule for enforcing a functional dependency.*

```
min(A,B,C)1                 <=> A =< B | C = A.
min(A,B,C)2                 <=> A >= B | C = B.
min(A,B,C)3                 ==> C =< A, C =< B.
min(A,B,C)5 \ min(A,B,D)4 <=> C = D.
```

For the constraint ***min(X,Y,Z)*** *if the upper bound of X changes (*ubc(X)*), only the occurrence in the first rule needs to be rechecked because of the guard $X \leq Y$. No other guard is affected by the* ubc(X) *event. Similarly, if variable X becomes fixed (*fixed(X)*) then only occurrences 4 and 5 from the last rule need to be rechecked,*[3] *since the guard (after normalisation) contains equality constraints. The resulting optimised implementation of* ***delay_min*** *is shown in Figure 8.2.*

Each solver event causes the execution of a specialised wakeup predicate which, in turn, rechecks occurrences linked to the guards possibly affected by the event. The wakeup predicate min_wakeup_1 *for event* ubc(X) *and* lbc(Y) *only rechecks occurrence* min_1 *from the first rule. Similarly, the wakeup predicate* min_wakeup_4_5 *for event* fixed(X) *or* fixed(Y) *only rechecks occurrences* min_4 *and* min_5*. Notice that for some events, such as* dc(Z)*, no occurrences ever need to be rechecked, since variable Z never appears in any of the guards.*

Using this version of ***delay_min*** *the execution of the goal*

```
[X,Y,Z] in 0..9, min(X,Y,Z), Z ≠ 2, Y ≤ 3, X ≥ 5.
```

would proceed as follows. The first constraint sets the domains of the variables are set to 0..9*. Next* ***min(X,Y,Z)*** *is executed, the delay goal set up, and each rule checked. Only the third rule fires adding the finite domain constraints $Z \leq X$ and $Z \leq Y$ into the built-in store, which does not change any domains. Now, when we add $Z \neq 2$ to the built-in store, the domain of Z changes. However, no guard mentions Z and therefore there is no delayed goal that wakes when Z's domain changes. When we add $Y \leq 3$ to the* ***fdint*** *store, only the upper bounds of Y and*

[3] We could use continuation optimisation to remove occurrence 5, however we will keep it in for the sake of this example.

```
delay_min(X,Y,Z,Id) :-
    delay(ubc(X),Id,min_wakeup_1(X,Y,Z,Id)),
    delay(ubc(Y),Id,min_wakeup_2(X,Y,Z,Id)),
    delay(lbc(X),Id,min_wakeup_2(X,Y,Z,Id)),
    delay(lbc(Y),Id,min_wakeup_1(X,Y,Z,Id)),
    delay(fixed(X),Id,min_wakeup_4_5(X,Y,Z,Id)),
    delay(fixed(Y),Id,min_wakeup_4_5(X,Y,Z,Id)).

min_wakeup_1(X,Y,Z,Id) :-
    min_1(X,Y,Z,Id).
min_wakeup_2(X,Y,Z,Id) :-
    min_2(X,Y,Z,Id).
min_wakeup_4_5(X,Y,Z,Id) :-
    min_4(X,Y,Z,Id),
    ( alive(Id) ->
        min_5(X,Y,Z,Id)
    ;   true).
```

Figure 8.2: Optimised compiled `min`/3 delay and wakeup handling code.

Z change. This causes the goal ***min_wakeup_2*** *to execute, which checks the second occurrence of* ***min*** *only. Similarly, when the constraint* $X \geq 5$ *is added to the built-in store, the lower bound of* X *changes and, again, only the second occurrence of* ***min*** *is checked. This time the rule fires, the CHR constraint is deleted and the constraint* $Z = Y$ *added to the store. This version makes a total of 7 occurrence checks: 5 for when the* ***min***/3 *constraint was first active, and then the second occurrence was checked twice during wakeup. Under the basic compilation, there would have been a total of 17 checks: 5 for the* ***min***/3 *constraint was first active, then 10 for when* Z *and* Y *change, and 2 for when* X *changes.* □

Unfortunately, specialising the delayed goals is unsafe with respect to the refined operational semantics. This is because we can no longer guarantee that occurrences will be checked in order after constraints are reactivated. This is best illustrated by a simple example.

Example 84 *Consider the* ***min*** *program from Example 83. If the upper bound of* X *changes, i.e. an* ***ubc(X)*** *event occurs, then the corresponding delayed goal only checks the first occurrence. Likewise, if the upper bound of* Y *changes (****ubc(Y)****), then the corresponding delayed goal only checks the second occurrence. Consider the constraint* ***min(X,Y,Z)*** *where the initial domains of each variable is* 0..9*, and suppose we introduce the constraint* $X + Y = 0$ *into the built-in store. The domains of* X *and* Y *reduce to the value* 0*. Hence, both the both the* ***ubc(X)*** *and* ***ubc(Y)*** *solver events occur.*

In HAL there is no way to control the order in which delayed goals are exe-

cuted. Therefore it is possible that `min_wakeup_2(X,Y,Z,Id)` *(associated with* `ubc(Y)`*) is called before* `min_wakeup_1(X,Y,Z,Id)` *(associated with* `ubc(X)`*). This means the second occurrence is checked before the first, which violates the refined semantics requirement that occurrences are checked in order.* □

It is important to note that the `min` program is confluent under the theoretical semantics, and therefore we always arrive at the same final state regardless of whether the refined semantics is adhered to. In practice it is very common that CHR programs that extend solvers rely on confluence under the theoretical semantics, as was discussed in Chapter 6. This makes this optimisation useful.

By default, the HAL CHR compiler assumes all programs are to be compiled with respect to the refined semantics. However, the user can enable the more specialised compilation by supplying an appropriate compiler flag.

8.3.3 Existential Variables

So far we have only dealt with solver events on variables that appear in the rule head, i.e. the non-existential variables. However, solver events associated to existential variables may also be required.

Example 85 *Consider the following rule from Example 40 where the guard contains an existential variable* `N` *introduced by guard normalisation.*

```
before_after(T1,D1,T2,D2) <=> exists [N] N = T1 + D1, N > T2 |
                                        T1 >= T2 + D2.
```

The first part of the guard is a tell constraint `N = T1 + D1`*, which binds* `N` *to be sum of* `T1` *and* `D1`*. Since the tell is deterministic (as function* `+` *is total), there are no associated solver events with this part of the guard. The second part of the guard* `N > T2` *is an ask constraint that needs to be rechecked on either* `lbc(N)` *or* `ubc(T2)` *solver events. The problem is that* `N` *is an existential variable, yet we need to recheck this rule on the* `lbc(N)` *solver event.* □

Ideally, we would want to map solver events on existential variables to solver events on non-existential variables. In Example 85 this is possible, since the only built-in constraint in the guard containing `N` is `N = T1 + D1`. Using knowledge about domain propagation, we know that an `lbc(N)` event must have been caused by either a `lbc(T1)` or `lbc(D1)` event. Thus, the complete minimal set of mapping solver events for the guard in this rule is `lbc(T1)`, `lbc(D1)` and `ubc(T2)`.

Unfortunately, mapping solver events on temporary variables to solver events on head variables requires intimate knowledge of the inner workings of the constraint solver in question. The current HAL CHR compiler uses a simpler approach: any solver event `event(`N`)` on temporary variable N may be caused by `changed(`X`)` for all non-existential variables $X \in vars(c)$, where c is a tell constraint containing N. Obviously, this solution is not as efficient, since we delay on `changed` solver events rather than more specialised ones. The result is a set of solver events that are not minimal.

Example 86 *Consider the rule in Example 85 with temporary variable* `N`*. We map the solver event* `lbc(N)` *to* `changed(T1)`*,* `changed(D1)` *based on the tell constraint* $N = T1 + D1$*. The final set of solver events is therefore* `changed(T1)`*,* `changed(D1)` *and* `ubc(T2)`*. Unlike before, the resulting solver events are not minimal.* □

8.4 Building Indexes on Solver Variables

In Chapter 7 we looked at efficient index structures for lookups, and showed that this dramatically improves the time performance of CHR programs. This is not surprising, since we can find matching constraints for rule heads much more efficiently. Up until now we have only considered indexes for ground constraints only. In this section we examine how to build efficient index structures on data involving solver variables, a task complicated by the need to take the solver state into consideration.

An index maintains a mapping from some key K (a tuple of solver terms) to some iterator of numbered CHR constraints matching that key, i.e., constraints containing the solver terms in K in the appropriate argument positions. A key point is that indexes implicitly *ask equality constraints*, therefore equating two solver terms may require an index to be updated. The following example makes this clearer.

Example 87 *Consider the indexes required for the following occurrences in the (normalised)* `leq` *program in Example 1.*

```
leq(X,Y)4, leq(Y1,X1)5 <=> X = X1, Y = Y1 | X = Y.
leq(X,Y)6, leq(Y1,Z)7  ==> Y = Y1         | leq(X,Z).
```

For the fourth occurrence, with active constraint `leq(A,B)` *we are looking for a CHR constraint of the form* `leq(B1,A1)` *to ensure the guard constraints* $A = A1, B = B1$ *hold. Hence, we need an index on both arguments of* `leq`*/2. Similarly, for the third occurrence. For the sixth occurrence, we need to find CHR constraints of the form* `leq(B1,_)` *to ensure that* $B = B1$ *holds. Thus, we need an index on the first argument. Similarly, for the seventh occurrence we need an index on the second argument.* □

The HAL CHR compiler supports several index structures: lists, trees and hash tables (see Section 7.2.3). We will concentrate here on trees where the most complex problems arise.

The core of the tree index code is an ordering $\preceq_B$ over solver terms according to the current state B of the built-in solver(s). This order is used to traverse the tree. We write $x \equiv_B y$ when $x \preceq_B y \wedge y \preceq_B x$ and $x \prec_B y$ when $x \preceq_B y \wedge \neg(y \preceq_B x)$. The ordering must be *total* and is assumed to satisfy the following *soundness* property: if $x \preceq_B y \wedge y \preceq_B x$ then $\mathcal{D} \models_{\mathcal{S}} B \rightarrow x = y$. Thus, the ordering answers whether equality constraints are *entailed* by the current store B. Note

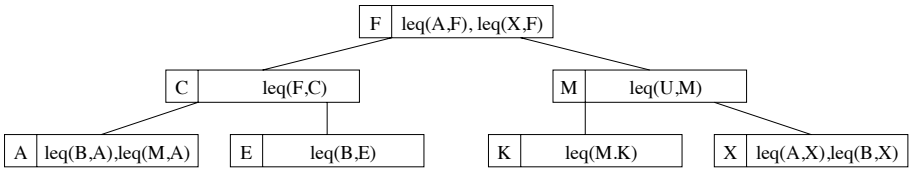

Figure 8.3: A tree of lists of CHR constraints indexed on the second argument

that it is not necessarily true that if $x \preceq_B y$ then $\mathcal{D} \models_{\mathcal{S}} B \rightarrow x \leq y$ (assuming the solver provides a $(\leq)/2$ constraint). In fact, it is preferable that the ordering only depends on the equality constraint, and this will become clear later.

For data that does not contain solver variables, the precedence relation $\preceq_B$ does not depend on the solver state B. As a result the relationship between two such terms in the ordering cannot change. However, when the value of the data does contain variables, the ordering relationship between two terms can change as the solver state evolves.

Example 88 *The usual ordering $\preceq_B$ of two Herbrand terms (Prolog's `@=<`) is defined as the lexicographic ordering (with variables before functors) of the solved form of the terms involved. In the empty solver state $B_0 = true$ we have $X \prec_{B_0} Y$, but when $B_1 \equiv (X = f(A))$ we have $Y \prec_{B_1} X$, but then adding another equation $B_2 \equiv (X = f(A) \wedge Y = f(B))$ we have $X \prec_{B_2} Y$, and finally adding $A = B$, $B_3 \equiv (X = f(A) \wedge Y = f(B) \wedge A = B)$ we have that $X \equiv_{B_3} Y$.* □

As illustrated above, during forward computation as the state B_0 changes to B_3 by adding new constraints to B_0, the order between solver terms can change in arbitrary ways. Therefore, a tree index based on comparison results which is correct at B_i, may become corrupted at B_{i+1}.

Example 89 *Figure 8.3 shows a binary search tree for `leq`/2 constraints indexed on the second argument based on the ordering $\preceq_{true}$ of an empty store. When we add the constraint $C = M$ the ordering $\preceq_{(C=M)}$ uses the solved form of the terms. Assuming M is replaced by C, then searching in the tree for entry with key K will not succeed. If C is replaced by M then the same applies to E.* □

Our solution is to repair tree indexes while they change, and this can be done as follows. Assume we have a tree index which is correct for state B_i and is corrupted for B_{i+1}. We delete all entries where $\preceq_{B_i}$ and $\preceq_{B_{i+1}}$ are not guaranteed to be the same, thus obtaining a correct tree. Then, the re-insertion can be made using the new ordering $\preceq_{B_{i+1}}$. There is however a slight problem: the deletion of the corrupted entries has to be performed *before* B_i actually changes into B_{i+1}, so that these entries can be correctly located.

Example 90 *Consider the tree in Figure 8.3 (which used order $\preceq_{true}$). Consider adding the constraint $C = M$ which replaces M by C. If when trying to locate the*

corrupted entry M we use the current *ordering $\preceq_{(C=M)}$, then at the root we will find $M \prec_{(C=M)} F$, go left and find $M \equiv_{(C=M)} C$. Unfortunately, we will have not discovered the M node, but the C node. In order to correctly locate the M node we need to use the previous ordering $\preceq_{true}$.* □

This problem is solved in HAL by requiring the solver to support two things. The first is a comparison predicate `compare(Result,X,Y)`[4] where `Result` is bound to either (`=`), (`<`) or (`>`) when $X \equiv_B Y$, $X \prec_B Y$ or $Y \prec_B X$ holds respectively, where B is the current solver state. The second, is a new special solver event `cc(X)` (*compare change*) which holds whenever a change in the solver state might cause the result of a comparison involving X to change. Importantly, unlike ordinary solver events, the `cc(X)` event fires and executes delayed goals "just before" the change in the solver state actually occurs.[5]

Another difference is that the `cc(X)` condition expects two goals instead of one: the first is executed just before the unification, and the other is executed immediately after it. To accommodate the need for two delayed goals, a special predicate `delay_update(`$Event$`,`Id`,`$Goal_1$`,`$Goal_2$`)` is used, where $Goal_1$ is the *before* goal, and $Goal_2$ is the *after* goal. Predicate `delay_update`/4 is otherwise exactly the same as `delay`/3.

We can use the `cc(X)` event and `delay_update` to implement safe tree indexing over solver terms.

Example 91 *The HAL Herbrand solver supports the* `cc(X)` *event. Thus, when using CHRs that extend the Herbrand solver we can setup delayed goals on this event to delete the modified constraints. The pseudo code shown in Figure 8.4 inserts a* `leq`/2 *constraint into the appropriate indexes, and sets up the necessary delayed goals for safe indexing. The predicates* `leq_temp_delete` *and* `leq_temp_insert` *respectively delete and insert the constraint from the tree indexes (for* `leq`/2*), and are woken up just before and just after unification.*

Returning to the problem of Example 90, when $C = M$ is added, just before the solver state (in this case the heap) is changed, the goal `leq_temp_delete(U,M,Id)` *is executed deleting the node with key M. Immediately after the unification, the goal* `leq_temp_insert(U,M,Id)` *is executed, which (re)inserts the constraint* `leq(U,M)` *into the correct position in the tree indexes.* □

Note that the issues we have discussed here are also similarly applicable to hash-based indexes over solver variables. The difference is that a special `hc(X)` (*hash change*) solver event is used instead of `cc(X)`.

If the built-in solver supports neither `cc(X)` nor `hc(X)` solver events then the current HAL compiler uses list indexes by default. The advantage of lists are that they do not depend on the solver state B. Unfortunately, as was shown in Section 7.4, list indexes are significantly slower than specialised index structures.

[4]The predicate `compare`/3 is analogous to the standard `compare`/3 predicate in Prolog.

[5]Attributed variables in most Prologs similarly must interrupt unification in order to perform computation just before the heap changes.

```
leq_insert(X,Y,Id) :-
    ( stored(Id) ->
        true
    ;   store(Id),
        Delete = leq_temp_delete(X,Y,Id),
        Insert = leq_temp_insert(X,Y,Id),
        delay_update(X,Id,Delete,Insert),
        delay_update(Y,Id,Delete,Insert),
        <insert-into-indexes>
    ).

leq_temp_delete(X,Y,Id) :-
    <delete-from-tree-indexes>.
leq_temp_insert(X,Y,Id) :-
    <insert-into-tree-indexes>.
```

Figure 8.4: Pseudo code for safe indexing over solver terms

8.5 Implementing Delay in HAL

Throughout this thesis we have taken the existence of dynamic scheduling for granted. The purpose of this section is to show how dynamic scheduling is actually implemented in HAL.

8.5.1 Fundamentals

It is the responsibility of each individual solver to implement their own version of the `delay`/3 predicate. This is achieved by providing an instance to the following typeclass.

```
:- class delay(Event) where [
     pred delay(Event,delay_id,pred),
     mode delay(oo,in,pred is semidet) is det
].
```

This defines a typeclass called `delay(Event)` where `Event` is the type of a set of solver events provided by the solver. Note that the mode of the solver event is `oo`, since solver events usually contain solver variables. The mode (`pred is semidet`) indicates that the third argument for `delay`/3 expects a `semidet` goal (i.e. the delayed goal is allowed to fail). The solver writer must ensure that calls to `delay`/3 never fail (hence `delay`/3 has `det` determinism). A call to `delay`/3 must never immediately call the goal, even if the solver event appears to be satisfied, e.g. delaying a goal on `fixed(3)`.[6]

[6]Older versions of `delay`/3, e.g. the version in [17], will immediately call the goal if the solver event is "satisfied". It was changed to give `delay`/3 a more consistent semantics, since

The first thing a solver writer must consider when implementing `delay` is what solver events need to be supported. Many factors must be taken into consideration, e.g. which solver events are easy to implement and which ask constraints are to be supported. Once this is decided, the solver writer defines the set of supported solver events by a *solver event type*, which is usually a discriminated union of each solver event.[7] For example, a finite domain solver defines the following solver event type.

```
:- typedef fd_event -> (fixed(fdint) ; lbc(fdint) ;
                                       ubc(fdint) ; dc(fdint)).
```

To assist the solver writer with implementing dynamic scheduling, there is a HAL library module that implements most of the essential functionality. The main feature the library provides is a *delayed goals collection* abstract data type, named `delayed_goals`, together with the following operations:

- `delayed_goals_init(`*Goals*`)` – binds *Goals* to a new empty `delayed_goals`;
- `add_delay(`*Goals*`,`*Id*`,`*Goal*`)` – adds a new delayed *Goal* and associated `delay_id` *Id* to `delayed_goals` *Goals*;
- `wake_up(`*Goals*`)` – Calls all goals in `delayed_goals` *Goals* whose associated *Id* has not been `kill`ed.

Notice that the interface to `add_delay`/3 is almost exactly the same as the interface to `delay`/3. The difference is that `add_delay`/3 is not overloaded, as it only operates on one type, namely `delayed_goals`, rather than solver events. Also, the *Goal* is only called during an explicit call to `wake_up(`*Goals*`)`, which we will discuss later.

Incidentally, the HAL library also provides the constraint identifier operations, i.e. `new(Id)`, `kill(Id)`, etc., defined in Section 4.3.2.

The solver writer implements `delay`/3 as follows. The `delay(`*Event*`,`*Id*`,`*Goal*`)` predicate maps *Event* to an associated `delayed_goals` for that event, then calls `add_delay`/3 with the same *Id* and *Goal*.

Example 92 *For example, the finite domain solver defines the* `delay`/3 *predicate for the* `fd_event` *solver event, as shown in Figure 8.5. Here, each function* `get_delayed_goals_`*event maps a* `fdint` *variable to a* `delayed_goals` *collection associated with that event. How this mapping is implemented is highly solver dependent, but usually the* `delayed_goals` *are stored somewhere in the internal data structure for solver variables. The code in Figure 8.5 has been slightly simplified since* `add_delay` *is typically not called when* `V` *is is an integer.* □

some solver events, e.g. `touched(X)`, have no sensible notion of "being satisfied".

[7]Nothing prevents solver events being any type, although the current CHR compiler only handles discriminated union types.

```
fd_delay(FDEvent,Id,Goal) :-
    ( FDEvent = fixed(V),
        FixedGoals = get_delayed_goals_fixed(V),
        add_delay(FixedGoals,Id,Goal)
    ; FDEvent = lbc(V),
        LBCGoals = get_delayed_goals_lbc(V),
        add_delay(LBCGoals,Id,Goal)
    ; FDEvent = ubc(V),
        UBCGoals = get_delayed_goals_ubc(V),
        add_delay(UBCGoals,Id,Goal)
    ; FDEvent = dc(V),
        DCGoals = get_delayed_goals_dc(V),
        add_delay(DCGoals,Id,Goal)
    ).
```

Figure 8.5: Example implementation of `fd_delay` for a finite domain solver

Next, the solver writer declares an appropriate instance of the `delay` typeclass. The instance declaration for `fd_delay` in Example 92 is the following.

```
:- instance delay(fd_event) where [
     pred(delay/3) is fd_delay
].
```

HAL will automatically replace (overloaded) calls to `delay`/3 to `fd_delay`/3 if the solver event type is `fd_event`.

The solver also must wakeup the appropriate delayed goals whenever a solver event actually occurs during a derivation. This is where the library predicate `wake_up(`*Goals*`)` is useful.

Example 93 *Figure 8.6 is an example of how the* `wake_up`/1 *predicate can be used to implement dynamic scheduling in a finite domain solver. Consider the constraint* `'=<_constant'` *which is similar to the standard* `=<` *constraint except the second argument is always a constant (i.e. an ordinary integer).*

The constraint works as follows: If the constant `C` *is smaller than the current upper bound for* `X` *(the return value of* `fd_max(X)`*), then we set the new upper bound for* `X` *to* `C`*. This is the purpose of the call to* `set_max(X,C)`*. We assume that* `set_max`/2 *also handles any additional bounds propagation resulting from changing the upper bound of* `X`*. After* `X` *has been assigned a new upper bound, we need to wake up delayed goals for the solver events* `fixed(X)`*,* `ubc(X)` *and* `dc(X)`*. First, we check if* `X` *has become fixed, i.e.,* `fd_min(X) = fd_max(X)`*, and if so we call* `wake_up`/1 *on the fixed goals of* `X`*. Like before, we use the special function* `get_delayed_goals_fixed(X)` *to retrieve these goals. Similarly, we call* `wake_up`/1 *on the delayed goals for* `ubc(X)` *and* `dc(X)`*.* □

```
'=<_constant'(X,C) :-
    UB = fd_max(X),              %% get current X upper bound
    ( C < UB ->                  %% change upper bound?
        set_max(X,C),            %% record new upper bound for X
        ( fd_min(X) = fd_max(X) ->
            wake_up(get_delayed_goals_fixed(X))
        ;   true )
        wake_up(get_delayed_goals_ubc(X)),
        wake_up(get_delayed_goals_dc(X))
    ;   true ).
```

Figure 8.6: Example usage of `wake_up`/1 to implement `delay`

Note that this implementation is rather naive since `wake_up`/1 may be called several times on the same event if further propagation results from `set_max`/2, e.g. if the upper bound of `X` changes twice. A more sophisticated solution accumulates a list of all solver events that have occurred, removes duplicates, then wakes up the delayed goals for each remaining event after bounds propagation has finished. This is the technique that the current version of the HAL bounds propagation solver uses.

Other standard operations on `delayed_goals` include

- `delayed_goals_reset(`*Goals*`)`: Destructively removes all delayed goals from *Goals* (thus effectively making *Goals* empty).
- `delayed_goals_merge(`*Goals*`1,`*Goals*`2,`*Goals*`3)`: Destructively merges collections *Goals*1 and *Goals*2 into *Goals*3. This is useful when combining the delayed goals of two variables after unification.

Note these operations are destructive (i.e. the old values of the inputs of these routines will be clobbered). This avoids the need to reinsert a new value back into the solvers internal data structures, however it does make the code *impure*.[8]

8.5.2 Polymorphic Solver Events

For CHR compilation, we require some solver events to be defined over multiple solvers with different solver types. One such solver event is `changed(X)`, which every solver in HAL supporting dynamic scheduling is encouraged to provide. The problem is that solver event `changed(X)` should work for any `X` of any solver type. This is in contrast with solver specific events, such as `lbc(X)`, etc., where `X` can only ever have one type, e.g. the type `fdint`. Our aim is to overload the constructor `changed`/1 so that it becomes a *polymorphic* solver event.

[8]Impure code is non-logical. See [40] for a description about impurity in Mercury, which is the same as in HAL.

In HAL, it is not possible to overload constructors directly.[9] Instead we overload a more specialised version of the `delay`/3 predicate, which is defined by the following typeclass.

```
:- class delay_changed(T) where [
    pred delay_changed(T,delay_id,pred).
    mode delay_changed(oo,in,pred is semidet) is det.
].
```

Notice the similarity between the `delay_changed`/1 and `delay`/1 typeclasses. Solvers provide instances to `delay_changed`/1 in much the same way as they provide instances to `delay`/1. The main difference is that the method `delay_changed`/3 expects a term of type `T` (i.e. the solver type), rather than a solver event type. In all of our examples up until this point, we should treat calls of the form `delay(changed(`X`),`Id`,`$Goal$`)` as `delay_changed(`X`,`Id`,`$Goal$`)`.

For many solvers, the solver event `changed(X)` is equivalent to an existing solver specific event. This makes implementing an instance for `delay_changed` trivial.

Example 94 *For example, the finite domain solver condition* ***`changed(X)`*** *is equivalent to the existing condition* ***`dc(X)`*** *(domain changed). We can therefore implement* ***`delay_changed`*** *as follows.*

```
fd_delay_changed(X,Id,Goal) :-
     delay(dc(X),Id,Goal).
```

□

8.5.3 Complex Solver Terms

Defining solver events is more difficult for solvers with *complex* solver terms. We define *simple* solver terms as variables, for fully ground data. For example, a solver term for the finite domain solver is always simple, since it is either a variable, or grounded to an integer. We define *complex* solver terms as a solver term that is not simple, i.e., has more elaborate instantiations. For example, consider a Herbrand solver over a list of finite domain variables, then $[1, X|Y]$ is a complex solver term, since it is neither a variable nor fully grounded. So far we have only considered solver events over simple solver terms, and in this section we show how to implement `delay_changed` over complex terms.

Usually, a solver event over a complex term can be rewritten into solver events over simple terms, by recursively descending through the term. This is the case with the `changed(X)` polymorphic solver event. We can express this as a simple term rewriting system consisting of the following rules

$$changed(f(X_1, ..., X_n)) \rightarrow changed(X_1) \vee ... \vee changed(X_n)$$

[9] It is possible to overload a function with the same name and arity as the constructor.

where f/n is a Herbrand constructor. Any of $changed(X_1), ..., changed(X_n)$ may be reduced further, until the only solver events remaining are on variables.

Example 95 *For example, we an rewrite the solver event* `changed(Ls)`*, where* `Ls` *is some list, using the rewrite rules*

$$\begin{array}{ll} changed([]) & \rightarrow \epsilon \\ changed([X|Y]) & \rightarrow changed(X) \vee changed(Y) \end{array}$$

The solver event `changed`$([1, X|Y])$ *is rewritten to* `changed`$(X) \vee$ `changed`(Y). □

Suppose the programmer declares the following Herbrand type, and wants to implement `delay_changed` over it.

```
:- typedef type(...) -> f1(type1,1,...,type1,n) ;
                        ⋮
                        fa(typea,1,...,typea,m) deriving delay.
```

We assume that all types $type_{i,j}$ also support `delay_changed`, or are not solver types. The pseudo code of `delay_changed` for this type is given in Figure 8.7. Lines (1)-(9) implement the instance method for `delay_changed`. Line (2) tests if `X` is bound (i.e. not a variable). If so, then in lines (3)-(8) we delay the *Goal* on any variables in any subterm of `X`. This involves deconstructing `X`, which we know is not a variable, then calling the overloaded `delay_changed` for each argument of `X`.

The other case is when `X` is an unbound Herbrand variable, which is handled by line (9). For Herbrand variables, `changed(X)` and (the non-overloaded) `touched(X)` solver events are very similar: both will wakeup the goal on any change to `X`, e.g. `X` unified with another variable, but after `X` has been bound to a non-variable, `touched(X)` will never again wakeup the goal. To overcome this difference, we delay a special *wrapper* goal named *type*`_changed` around the original *Goal* and then delay it on the `touched(X)` solver event. The wrapper goal, defined on lines (10)-(14), calls *Goal* on line (11) (since if the wrapper goal is called, then `X` must have changed), then checks if `X` has been bound on line (12). If `X` has been bound, it means that the wrapper goal (which was delayed on the `touched(X)` event) will no longer be called. To correctly implement the behaviour of `changed(X)`, we (recursively) call *type*`_delay_changed` again with the same arguments. This recursive call will do the necessary deconstruction, and set up delayed goals on any subterms of `X`.

Example 96 *Consider a Herbrand solver over lists of finite domain integers, and consider that we need to implement the* `changed(Ls)` *polymorphic solver event. This is implemented in Figure 8.8.* □

We can immediately see from the pseudo code and examples that the number of actual delayed goals (i.e. calls to `delay`/3) is equal to the number of variables in solver term `X`. Each call to `delay`/3 consumes memory, since the goal and the

```
type_delay_changed(X,Id,Goal) :-                                    (1)
    ( nonvar(X) ->                                                  (2)
        ( X = f_1(Y_1,...,Y_n),                                     (3)
            delay_changed(Y_1,Id,Goal),                             (4)
            ⋮
            delay_changed(Y_n,Id,Goal)                              (5)
        ;
            ⋮
        ; X = f_a(Z_1,...,Z_m),                                     (6)
            delay_changed(Z_1,Id,Goal),                             (7)
            ⋮
            delay_changed(Z_m,Id,Goal)                              (8)
        )
    ;   delay(touched(X),Id,type_changed(X,Id,Goal))                (9)
    ).

type_changed(X,Id,Goal) :-                                          (10)
    call(Goal),                                                     (11)
    ( nonvar(X) ->                                                  (12)
        type_delay_changed(X,Id,Goal)                               (13)
    ;   true                                                        (14)
    ).
```

Figure 8.7: Pseudo code for `delay_changed` over a complex type

identifier must be stored into a `delayed_goals` data structure. If `X` contains a large number of variables, then this may be inefficient. Also, if `X` contains repeats of the same variable `Y`, then current implementation will set up a delayed goal for every instance of `Y` in `X`. Again, this may be inefficient, since the *Goal* may be called more times than is necessary.

8.5.4 Index Related Dynamic Scheduling

The implementation of `delay_update`/4 is very similar to the other types of dynamic scheduling. The main differences are that `delay_update`/4 expects two goals: one executed immediately before the solver event, and one executed immediately after it. Two polymorphic solver events are supported, namely `cc(X)` and `hc(X)` (see Section 8.4), and their corresponding typeclass definitions are shown in Figure 8.9. Notice that the interfaces for the overloaded predicates `delay_update_cc`/4 and `delay_update_hc`/4 are exactly the same.

The implementation of `delay_update` is very similar to the other forms of dynamic scheduling in HAL.

```
list_delay_changed(Ls,Id,Goal) :-
    ( nonvar(Ls) ->
        ( Ls = [],
            true
        ; Ls = [X|Ls1],
            delay_changed(X,Id,Goal),
            delay_changed(Ls1,Id,Goal)
        )
    ;   delay(touched(Ls),Id,list_changed(Ls,Id,Goal))
    ).

list_changed(Ls,Id,Goal) :-
    call(Goal),
    ( nonvar(Ls) ->
        delay_changed(Ls,Id,Goal)
    ;   true
    ).
```

Figure 8.8: Implementation of `delay_changed` over a list solver

```
:- class delay_update_cc(T) where [
    pred delay_update_cc(T,delay_id,pred,pred),
    mode delay_update_cc(oo,in,pred is semidet,pred is semidet)
].

:- class delay_update_hc(T) where [
    pred delay_update_hc(T,delay_id,pred,pred),
    mode delay_update_hc(oo,in,pred is semidet,pred is semidet)
].
```

Figure 8.9: The `delay_update` typeclasses for indexing

Example 97 *Consider the implementation of* `delay_update` *for a Boolean solver. The* `instance` *declaration and matching predicate is shown in Figure 8.10. We assume that the type of a Boolean variable is* `bool`*. Notice that two* `delayed_goals` *collections are required (one for the before and after goals).*

Consider a simple Boolean constraint `true(X)` *which holds if variable* `X` *is true. The code for* `true(X)` *is shown in Figure 8.11. Predicate* `set_to_true(X)` *does the actual binding of variable* `X` *to the value true. Immediately before the binding takes place, the* before goals *for condition* `cc(X)` *are called. After the binding, the corresponding* after goals *are called. The call* `propagate(X)` *propagates the new value for* `X`*. Note that this propagation happens after the goals have*

```
:- instance delay_update_cc(bool) where [
    pred(delay_update_cc/4) is bool_delay_update_cc
].

bool_delay_update_cc(X,Id,Before,After) :-
    BeforeGoals = get_delayed_goals_before_cc(X),
    add_delay(BeforeGoals,Id,Before),
    AfterGoals = get_delayed_goals_after_cc(X),
    add_delay(AfterGoals,Id,After).
```

Figure 8.10: Implementation of `delay_update_cc` for a Boolean solver

```
true(X) :-
    ( nonvar(X) ->
        test_is_true(X)
    ;   BeforeGoals = get_delayed_goals_before_cc(X),
        wake_up(BeforeGoals),
        set_to_true(X),
        AfterGoals = get_delayed_goals_after_cc(X),
        wake_up(AfterGoals),
        propagate(X)
    ).
```

Figure 8.11: Implementation of the `true(X)` constraint supporting `delay_update`

been called, otherwise we may wakeup goals from `delay`*/3, and hence call CHR constraints before they are added back into the store.* □

For many solvers, the `hc(X)` solver event is equivalent to `cc(X)`. This means that if `cc(X)` is already implemented, then to implement `hc(X)` we just need to provide an appropriate `instance` declaration. For example, assuming that `cc(X)` and `hc(X)` are equivalent for the Boolean solver in Example 97, we just need to provide the following instance declaration to support `hc(X)`.

```
:- instance delay_update_hc(bool) where [
    pred(delay_update_hc/4) is bool_delay_update_cc
].
```

For solvers with complex types, the implementation of the `delay_update` predicates is very similar to `delay_changed` in Section 8.5.2. For Herbrand solver types, we recursively descend through the term using the following rule.

$$cc(f(X_1, ..., X_n)) \rightarrow cc(X_1) \vee ... \vee cc(X_n)$$

The only slight complication is that only the after goal needs to be wrapped (as in lines (10)-(14) in Figure 8.7).

8.6 Experimental Results

In this section we show the benefit of CHR optimisation on several example programs and benchmarks. The HAL CHR compiler has been extended to support `asks` declarations, to translate guard constraints to ask constraints, and to use solver events to set up minimal re-execution when a solver changes. It also automatically builds appropriate index structures over solver variables for the joins required by CHR rules. Both balanced trees and hash table index structures are supported.

For the experiments, we will use the following test suite:

- `fib` – The inefficient version of Fibonacci program from Example 16 (where rules `f2` and `f3` are swapped). Benchmark `fib(`n`)` calculates the n^{th} Fibonacci number.

- `min` – The `min` program from Example 37. Benchmark `path(`n`,`m`)` attempts to assign weights to a directed graph of m nodes such that there exists a shortest path equal to n. Note that `path(5,1000)` has one solution.

- `leq` – The classic less-than-or-equal-to program from Example 1. Benchmark `leq(`n`)` computes

$$X_0 \leq X_1 \leq ... \leq X_{n-1} \leq X_n \leq X_0$$

  which results in the unification of all X_i.

- `leq_bounds` – same as `leq` but extends a bounds propagation solver.

- `boolean` – A Boolean solver with rules similar to that as shown in Example 81. Benchmark `queens(`n`)` finds a solution for the classic n-queens problem using Boolean constraints.

- `queue` – Same as the `queue` benchmark in Section 7.4, except this version extends a Herbrand solver. Benchmark `queue(`n`)` adds n elements, then retrieves the same n elements.

- `bounds` – Similar to the `bounds` program in Appendix A.1, except bounds variables are Herbrand variables, and equality is implicitly handled by the Herbrand solver (rather than by a `eq`/2 CHR constraint and an explicit rule).

- `chameleon` – Rules generated by the Chameleon programming language [81] for resolving typeclass functional dependencies [25]. Benchmark `fd(`n`)` resolves n class constraints with functional dependencies.

Table 8.1: Statistics from each of the example programs extending solvers

Prog.	Extends	$\|c\|$	<=>	\	==>	$\|r\|$
fib	bounds	1	1	1	1	3
min	bounds	4	4	2	1	7
leq	herbrand	1	2	1	1	4
leq_bounds	bounds	1	2	1	1	4
boolean	herbrand	11	3	24	0	27
queue	herbrand	5	5	2	0	7
bounds	herbrand	12	8	5	5	18
chameleon	herbrand	1	9	0	1	10
list	both	3	7	0	1	8

- `list` – Defines `list` constraints, namely `length`/2 (as shown in Example 32), `append`/3 and `neq`/2 (list disequality). Benchmark `triples(`n`,`m`)` tries to find n triples of sequences (lists) of positive integers $\leq m$ such that (1) the length of each sequence is $\leq n$ (2) each sequence is not equal to any other sequence (from any triple); and (3) the concatenation of elements for all triples must be equal.

Table 8.1 summaries relevant information about each program. The *Extends* column identifies which solver(s) the program extends. Here '`both`' means the program extends both the Herbrand and finite domain solvers. The $|c|$ column is the number of CHR constraints defined by each program. Next the <=> column is the number of simplification rules, \ is the number of simpagation rules, and ==> is the number of propagation rules. Finally, the $|r|$ column is the total number of rules.

The results are shown in Table 8.2 and Table 8.3 respectively. All timings are the average over 10 runs on a 1.2GHz AMD Athlon Processor with 1.5Gb of RAM running under Linux (Debian) with kernel version 2.4.22 and are given in milliseconds. SICStus Prolog 3.8.6 is run under compact code (no fastcode for Linux). We compare to SICStus CHRs where possible just to illustrate that the HAL CHR implementation is mature and competitive.

Table 8.2 shows the benefit of using index structures over solver variables. We test all programs except `list`, which does not require indexing since all rules are single-headed. Here, all other optimisations are enabled, including accurate specialisation under the assumption of ω_t confluence. The *list*, $+tree$ and $+hash$ versions use a `list`, `tree` and `hash` indexes respectively. As was the case in Section 7.4, both `tree` and `hash` indexes are superior over a `list` index overall, with a 80% improvement for lists, and an 88% improvement for hash tables. The exception is the `boolean` program, which showed that a `list` index is faster in some cases. This occurs when the additional overhead of creating and maintaining more complicated indexes (e.g. setting up delayed goals on the `cc(`X`)` and `hc(`X`)`

Table 8.2: Execution times (ms) for various benchmarks testing indexing over solver variables

Prog.	Benchmark	*list*	*+tree*	*+hash*	*SICS*
`fib`	`fib(25)`	2239	2072	1811	n/a
`min`	`path(5,1000)`	4127	3156	3455	n/a
`leq`	`leq(80)`	8162	1831	1064	4064
`leq_bounds`	`leq(80)`	5339	636	600	n/a
`boolean`	`queens(12)`	141	236	165	3176
`boolean`	`queens(21)`	19123	21628	16116	235661
`queue`	`queue(5000)`	4997	43	41	6612
`bounds`	`queens(8)`	3706	425	142	–
`bounds`	`queens(15)`	417123	5189	1606	–
`chameleon`	`fd(2000,10)`	583	47	16	4277
`chameleon`	`fd(2000,2000)`	4244	2306	979	–
	geom. mean	4626	20%	12%	422%*

conditions) outweighs the benefit of faster lookups. This is especially true if the CHR store is relatively small, as is the case with the `boolean` program and these benchmarks.

Finally, *SICS* is provided to show how our CHR compiler compares with other existing implementations. Note that the `fib`, `min` and `leq_bounds` cannot be compiled to SICStus CHR, since only a Herbrand solver can be extended. The `bounds` program takes too long to complete the benchmarks, since the SICStus CHR compiler lacks join ordering, hence resulting in a very inefficient executable. Interestingly, the `chameleon` program with the `fd(2000,2000)` benchmark also takes too long to terminate. This is because of the order rules are rechecked when the constraints wakeup. This is discussed further below.

Table 8.3 shows the benefit of using specialisation on various benchmarks. Here, all other optimisations are enabled, and each program uses hash indexes. The $-$*spec* version uses no specialisation, and all occurrences are rechecked on the `changed(`X`)` event for any variable X in the CHR constraint. The $+1/2$*spec* version enables the specialisation described in Section 8.3.1, where `changed` events are either removed or replaced with a more specific event based on the analysis of the guards if possible. Finally, the $+$*spec* version enables full specialisation as described in Section 8.3.2 under the assumption that each program is ω_t confluent. Note that the $+$*spec* version is equivalent to the *hash* version in Table 8.2.

Overall we see that both kinds of specialisation are superior, with a 12% improvement for $+1/2$*spec*, and a 51% improvement for $+$*spec*. Most notably, the $+$*spec* version of `chameleon` is significantly faster than the $+1/2$*spec* version. The reason is because of the following two rules.

```
fd(A,B), fd(A,C) ==> B=C.
```

Table 8.3: Execution times (ms) for various benchmarks testing specialisation

Prog.	Benchmark	$-spec$	$+1/2spec$	$+spec$
`fib`	`fib(25)`	1929	1679	1811
`min`	`path(5,1000)`	5683	4736	3458
`leq`	`leq(80)`	1064	1064	1064
`leq_bounds`	`leq(80)`	611	611	600
`boolean`	`queens(12)`	218	161	165
`boolean`	`queens(21)`	19322	15404	16116
`bounds`	`queens(15)`	1632	1606	=
`chameleon`	`fd(2000,10)`	4492	4198	16
`list`	`triples(5,2)`	5622	4847	4663
`list`	`triples(5,3)`	23729	20330	19594
	geom. mean	2798	88%	49%

```
fd([A],B) <=> B = [C], fd(A,C).
```

With the $+1/2spec$, the propagation rule is always checked first when a `fd`/2 constraint wakes up, whereas the only the simplification rule is checked in the $+spec$ version. It is significantly faster to check the simplification rule compared to the propagation rule, hence the big difference in the performance between the $+1/2spec$ and $+spec$ versions. This also explains why the $SICS$ version performs poorly in Table 8.2.

On the other hand, the $+spec$ version is sometimes worse than the $+1/2spec$ version. This can occur is woken up constraints are deleted by the first few rules that are checked, thus the benefit of more accurate specialisation is lost. Since accurate specialisation usually requires more delayed goals to be created, the overhead costs are greater, and hence the slowdown. This is the case with the `boolean` program, where Boolean constraints are often quickly simplified once an argument becomes more bound. Another potential problem with accurate specialisation is that a computationally expensive occurrence may be visited before a cheaper one (this is the opposite to the what is happening with the `chameleon` example). We did not observe this in any of the benchmarks, however it is a possibility the programmer should be aware of.

8.7 Summary

In this chapter we have shown how we can extend arbitrary solvers supporting dynamic scheduling using CHRs rules, from both a theoretical and practical point of view.

We presented a formalisation of the implicit wakeup policy used by the basic compilation. This involved formalising exactly what (the naive version of)

`delay`/3 means, and then showing it satisfies the conditions in Definition 10.

We investigated how to compile the execution of CHR rules so that we re-execute the minimum number of rules on a change of solver state. Experiments show that specialising the re-execution is beneficial, and that the more expensive the joins in the CHR rules the more beneficial it is. To fully specialise the wakeup code the program needs to be confluent under the theoretical operational semantics, which is the case with all the examples in this chapter.

The use of efficient indexes is vital to support efficient execution of CHRs, and hence we need to support such indexes over changing solver data. We presented a solution to this problem by using a special variant of dynamic scheduling: one where delayed goals are executed immediately before, and immediately after a change in some solver data occurs. We show that despite this additional overhead, using indexes is still significantly faster than the safe alternative, which is to use list indexes.

We explained in detail how dynamic scheduling is implemented in HAL, and how solver writers can implement the dynamic scheduling interface, so solvers can become extensible. We showed that much of the functionality of dynamic scheduling is implemented in a special library module, which the solver writer can take advantage of. Several other topics were taken into consideration, such as implementing solver events over complex terms, and implementing the special form of dynamic scheduling for indexing.

Chapter 9

Conclusion

9.1 Summary and Conclusions

In this thesis we presented the theory and practice of CHR compilation.

Chapter 3 presented the first formalisation of the refined operational semantics for CHRs, which is the operational semantics used by most current CHR implementations. This work therefore lays the theoretical foundation of modern CHR compilers and interpreters. The formal definition is used to prove correctness of the refined semantics with respect to the theoretical semantics. Results linking termination and confluence in the theoretical semantics to the refined semantics was also shown.

The practical aspects of basic CHR compilation were introduced in Chapter 4. It presented a minimal compiler and runtime environment which implement the refined operational semantics. The target is a CLP language such as Prolog or HAL. The compilation of a restricted form of guards with existential variables was also covered, however this assumes additional information provided by the solver in the form of `asks` declarations.

Chapter 5 presented a program analysis framework for CHRs based on abstract interpretation, and provided two important instances of the framework: late storage analysis and functional dependency analysis. The analysis framework relies on the call-based operational semantics for CHRs, which is a different formulation of the refined semantics designed to make analysis easier. The very fact that program analysis is even possible strengthens the case for the refined semantics. For example, if functional dependency analysis were to be adapted to the theoretical semantics, the resulting analysis will be too far weak to be useful.

One application of functional dependency analysis was a confluence test under the refined semantics, which was covered in Chapter 6. Functional dependency information is used to decide if a given rule is matching complete, i.e., will try all of its potential matches at runtime. Matching completeness or independence guarantees confluence under some other assumptions, such as termination, groundness and order independence. The usefulness of the confluence was shown by the fact that it detected a bug in an early version of the `bounds` program. Currently, the

confluence test is fairly weak, but could potentially be strengthened by taking into account standard CHR idioms.

Chapter 7 discussed several CHR optimisations. These ranged from simple peephole optimisations, such as removal of propagation histories, delay avoidance, etc., to more high-level optimisations based on the program analysis, such as functional dependencies used for join ordering and index selection. Optimisation is critical for reasonable performance for CHR programs, as was demonstrated by the experimental results.

Finally, Chapter 8 examined the issues that emerge from CHR programs extending underlying solver(s). This includes some theoretical results, which showed that the set of CHR constraints woken up by delaying on `changed`/1 satisfies the definition of a *wakeup policy*. We also covered some additional optimisations, which were wakeup specialisation and indexes over solver terms. Both of these rely on the dynamic scheduling interface to the underlying solver, including `asks` declarations and ask constraints. We also presented a guide to implementing this interface. Most of the time, the interface can be implemented by adapting the pseudo code presented in this chapter.

9.2 Contributions

Here we list the benefits of this thesis for CHR programmers, CHR compiler writers and CHR theoreticians.

Benefits for CHR programmers

The main benefit of this work is the considerable improvement and modernisation of the compilation of CHRs over earlier versions.

Most of this benefit is a direct result from the formalisation of the refined semantics, since this leads to a better understand of how CHR programs actually behave. A programmer can take advantage of this, and write CHR programs that rely on the refined semantics. In fact, most of the CHR programs in this thesis rely on the refined semantics. This represents a reinvention of CHRs into a more generic (yet powerful) rule based language that still retains some of the declarative aspects of the theoretical semantics. For example, the `bounds` program from Example 69 relies on the refined semantics, yet the programmer could add "declarative" rules (based on the mathematical properties of the constraints), e.g. the transitivity rule.

```
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

The formalisation of the refined semantics led to an abstract interpretation framework for CHRs, which has been used for discovering information about late storage and functional dependencies. The results of the analysis can be used to help detect non-confluence, and to optimise CHR programs. Detecting non-confluence is clearly beneficial, since non-confluence usually indicates a bug in

the program. The confluence test has successfully detected a bug in the `bounds` program.

Optimising CHRs is also clearly beneficial. The global optimisations of join ordering and early guard scheduling are essential for programs with large rule heads, and using efficient index structures is also of significant benefit. Local optimisations, i.e. continuation, lateness and never-stored optimisation, provide additional improvement. If the program extends a built-in solver, then specialisation and indexing over solver variables is also useful. For some simple programs, e.g. the `gcd` program, the resulting optimised executable is very competitive compared with a hand implemented version.

Benefits to CHR compiler writers

Chapters 4, 7 and 8 can be thought of as a guide to building an advanced CHR compiler in a programming language such as HAL. A majority of the ideas presented in these chapters could easily be extended to other programming languages as well. The obvious benefit is that CHR compilation has now been extensively documented.

One benefit is that the programmer can decide how advanced to make the CHR compiler. For a minimal, yet fully functional, CHR compiler, Chapter 4 describes a basic compiler and runtime environment that implements the refined operational semantics. For a more advanced compiler, Chapter 7 for how to optimise CHRs. Chapter 8 describes some additional optimisations for CHRs that extend another solver.

Another benefit is that the refined semantics is a de facto standard for CHR compilation. A CHR compiler writer may wish to try different approaches to compiling CHRs, yet still be based on the refined semantics.

Benefits to CHR theoreticians

The refined semantics is a new formalisation of CHRs, which therefore opens up more opportunities for CHR theoreticians. The confluence and termination of CHRs under the refined semantics is an interesting (and relatively unexplored) area of potential research.

9.3 Future Work

We briefly discuss some ideas for future work directly related to this thesis.

Analysis and optimisation

There is considerable scope for more research into the analysis and optimisation of CHRs. For example, if consecutive rules share similar heads, then perhaps the matching algorithms can be combined, hence avoiding redundant work.

We also intend to further refine and formalise the CHR analysis framework and instances of the framework.

Language issues

The refined semantics is still nondeterministic, and it might be worthwhile further "refining" the semantics until it is deterministic. This may lead to more accurate analysis and more optimisations. It would also make all CHR programs trivially confluent, although this does nothing to solve matching completeness related bugs.

Also, alternatives to the refined semantics itself have not been fully explored. Perhaps there are modifications to the semantics which can lead to better optimisation. For example, removing the woken up constraints from the store during a **Solve** transition both simplifies the implementation (in HAL) and means that late storage is applicable to the (reactivated) constraints. We intend to explore these possibilities in the future.

How to program

If the refined semantics stands the test of time, then the issue of *how* to program in CHRs needs to be throughly explored. This thesis already provides some hints. For example, the confluence test and some optimisations rely on functional dependency information, and the analysis can only detect this from rules of a particular form, e.g.

```
p(X,Y) \ p(X,Z) <=> true.
```

Therefore, if any CHR constraint has an inherent functional dependency, it is good practise for the programmer to write down such a rule explicitly.

Bibliography

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, LNCS 1330, pages 252–266. Springer-Verlag, 1997.

[2] S. Abdennadher. Constraint Handling Rules: Applications and Extensions. In *Proceedings of 2nd International Workshop on Optimization and Simulation of Complex Industrial Systems. Extensions and Applications of Constraint-Logic Programming and 7th International Workshop on Deductive Databases and Logic Programming in conjunction with the 12th International Conference on Applications of Prolog*, Tokyo, 1999. Invited Talk.

[3] S. Abdennadher and T. Frühwirth. On Completion of Constraint Handling Rules. In *4th International Conference on Principles and Practice of Constraint Programming*, LNCS 1520, pages 25–39. Springer-Verlag, 1998.

[4] S. Abdennadher, T. Frühwirth, and H. Muess. Confluence and Semantics of Constraint Simplification Rules. *Constraints*, 4(2):133–166, 1999.

[5] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A java constraint kit. In *Electronic Notes in Theoretical Computer Science*, volume 64, 2002.

[6] H. Aït-Kaci. *Warren's Abstract Machine*. MIT Press, 1991.

[7] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[8] K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, LNCS 1713, pages 58–72. Springer Verlag, 1999.

[9] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge Univ. Press, 1998.

[10] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, third edition, 2000.

[11] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The ciao prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.

[12] A. Cheadle, W. Harvey, A. Sadler, J. Schimpf, K. Shen, and M. Wallace. ECL^{i}PS^e: An Introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College London, 2003.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87, Berlin, June 2003. Springer.

[14] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer, fourth edition, 1994.

[15] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. YAP User's Manual. http://www.ncc.up.pt/∼vsc/Yap/, 2005.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[17] M. García de la Banda, B. Demoen, K. Marriott, and P. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, LNCS 2441, pages 47–66. Springer-Verlag, 2002.

[18] M. Garcia de la Banda, D. Jeffery, K. Marriott, P. Stuckey, N. Nethercote, and C. Holzbaur. Building constraint solvers with HAL. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS 2237, pages 90–104. Springer-Verlag, 2001.

[19] B. Demoen. http://www.cs.kuleuven.ac.be/∼bmd/hProlog/, 2005.

[20] B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practices of Constraint Programming*, LNCS 1713, pages 174–188. Springer-Verlag, 1999.

[21] B. Demoen, M. Gracia de la Banda, W. Harvey, K. Marriot, P. Schachte, and P. Stuckey. Compiling the HAL variable to Mercury. Technical Report CW273, Department of Computer Science, K.U.Leuven, 1998.

[22] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In *International Conference on Logic Programming*, pages 774–790. MIT Press, 1993.

[23] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The Constraint Logic Programming Language CHIP. In *Proceedings of the Second International Conference on Fifth Generation Computer Systems*, pages 693–702. Ohmsha Publishers, 1988.

[24] G. Duck, M. Garcia de la Banda, and P. Stuckey. Compiling ask constraints. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS 3132, pages 105–119. Springer-Verlag, September 2004.

[25] G. Duck, S. Peyton-Jones, P. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proceedings of European Symposium on Programming*, LNCS 2986, pages 49–63. Springer-Verlag, 2004.

[26] G. Duck, P. Stuckey, M. Garcia de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 79–90. ACM Press, 2003.

[27] G. Duck, P. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS 3132, pages 90–104. Springer-Verlag, September 2004.

[28] C. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July 1981.

[29] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[30] T. Frühwirth. Constraint simplification rules. Technical Report LP-63, ECRC Munich, Germany, October 1991.

[31] T. Frühwirth. Constraint simplification rules. Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992.

[32] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.

[33] T. Frühwirth. Proving termination of constraint solver programs. In *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop*, LNCS 1865, pages 298–317. Springer-Verlag, 1999.

[34] T. Frühwirth and S. Abdennadher. The Munich Rent Advisor: A Success for Logic Programming on the Internet. *Theory and Practice of Logic Programming*, 1(3):303–319, 2001.

[35] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, Berlin, 2003.

[36] T. Frühwirth and P. Brisset. High-level implementations of constraint handling rules. Technical Report ECRC-95-20, ECRC Munich, Germany, 1995.

[37] M. García de la Banda, P. Stuckey, W. Harvey, and K. Marriott. Mode checking in HAL. In J. LLoyd et al., editor, *Proceedings of the First International Conference on Computational Logic*, LNCS 1861, pages 1270–1284. Springer-Verlag, July 2000.

[38] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[39] Programming Systems Group. *SICStus Prolog User's Manual*, release 3#0 edition, 1995.

[40] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, the University of Melbourne, 1996. Available from http://www.cs.mu.oz.au/mercury.

[41] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, 1996.

[42] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, San Francisco, CA, 1999.

[43] B. Herbig. Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismum (in German). Master's thesis, University of Kaiserslautern, Germany, April 1993.

[44] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, LNCS 631, pages 260–268. Springer-Verlag, 1992.

[45] C. Holzbaur, M. Garcia de la Banda, P. Stuckey, and G. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules*, 2005. To appear.

[46] C. Holzbaur and T. Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, LNCS 1702, pages 117–133. Springer-Verlag, 1999.

[47] C. Holzbaur and T. Frühwirth. A Prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence*, 14(4), 2000.

[48] C. Holzbaur, P. Stuckey, M. Garcia de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS 2237, pages 74–89. Springer-Verlag, 2001.

[49] A. Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[50] ISO. Standard for the programming language Prolog. Standard Number ISO/IEC 12311-1:1995, 1995.

[51] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. Fourteenth ACM Symp. Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[52] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20, 1994.

[53] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 4(3):339–395, 1992.

[54] Java programming language. http://java.sun.com, 2005.

[55] JACK: Java constraint kit. http://www.pms.informatik.uni-muenchen.de/software/jack/index.html, 2002.

[56] S. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999. Available at http://www.haskell.org.

[57] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[58] A. Macdonald, P. Stuckey, and R.Yap. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1993 International Symposium*, pages 75–93, Vancouver, Canada, October 1993. MIT Press.

[59] M. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *International Conference on Logic Programming*, pages 858–876. MIT Press, 1987.

[60] K. Marriott, H. Søndergaard, and N. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[61] K. Marriott and P. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[62] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall, London, fourth edition, 1997.

[63] D. Miranker. TREAT:A Better Match Algorithm for AI Production Systems. In *National Conference on Artificial Intelligence*, pages 42–47. Morgan Kaufmann, August 1987.

[64] M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[65] R. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.

[66] D. Overton. *Precise and Expressive Mode Systems for Typed Logic Programming Languages*. PhD thesis, The University of Melbourne, 2003.

[67] Quintus Prolog Manual. http://www.sics.se/isl/quintus/html/quintus, 2003.

[68] K. Sagonas, T. Swift, D. Warren, J. Freire, and P. Rao. The XSB System Version 2.2 Programmer's Manual. http://www.cs.sunysb.edu/∼sbprolog/xsb-page.html, 2005.

[69] V. Saraswat. *Concurrent Constraint Programming Languages*. MIT Press, 1993.

[70] T. Schrijvers. http://www.cs.kuleuven.ac.be/∼dtai/projects/CHR/, 2005.

[71] T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *First workshop on constraint handling rules: selected contributions*, 2004. Published as technical report: Ulmer Informatik-Berichte Nr. 2004-01, ISSN 0939-5091, http://www/informatik.uni-ulm.de/epin/pw/10481.

[72] T. Schrijvers and T. Frühwirth. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Department of Computer Science, July 2004.

[73] T. Schrijvers, P. Stuckey, and G. Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, 2005. (to appear).

[74] T. Schrijvers and D. Warren. Constraint handling rules and tabled execution. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS 3132, pages 120–136. Springer-Verlag, September 2004.

[75] T. Schrijvers, J. Wielemaker, and B. Demoen. Constraint Handling Rules for SWI-Prolog. In *Workshop on (Constraint) Logic Programming, Ulm*, 2005.

[76] G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 405–420. MIT Press, London, 1993.

[77] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.

[78] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.

[79] L. Sterling and E. Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[80] P. Stuckey and M. Sulzmann. A Theory of Overloading. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 167–178. ACM Press, 2002.

[81] P. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon System. In *First workshop on constraint handling rules: selected contributions*, 2004. Published as technical report: Ulmer Informatik-Berichte Nr. 2004-01, ISSN 0939-5091, http://www/informatik.uni-ulm.de/epin/pw/10481.

[82] A. Taylor. PARMA–bridging the performance gap between imperative and logic programming. *Journal of Logic Programming*, 29(1–3), 1996.

[83] The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide. http://www.haskell.org/ghc/, 2005.

[84] M. Thielscher. FLUX: A Logic Programming Method for Reasoning Agents. *Theory and Practice of Logic Programming*, 2004.

[85] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[86] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proceedings of the Fourteenth ACM Symposium Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[87] D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

[88] J. Wielemaker. SWI-Prolog Home Page. http://www.swi-prolog.org/, 2005.

[89] R. Yap. *The Design, Programming Methodology, and Implementation of CLP(R)*. PhD thesis, Dept. of Computer Science, Monash University, Australia, April 1995.

Appendix A

Example Programs

A.1 Bounds Propagation Solver in HAL CHR

The following is the full source code for a bounds propagation solver implemented in HAL CHR. It provides an internal representation of solver variables (type `fdint` and constraint `bounds`/3), and finite domain constraints `eq` (=), `geq` (≥), `neq` (≠) and `plus` (+).

```
:- module bounds.

:- import int.
:- import chr.

:- typedef fdint -> fdint(int).

%%% Auxiliary constraint for the next free variable identifier.
%%% This is an example of using CHR constraints as global variables.
:- chrc next_id(int).
:- mode next_id(in) is det.
next_id(_) \ next_id(_) <=> true.

%%% Creates a fresh fdint variable with initial bounds -1000..1000.
%%% Also initialises next_id/1 when first called.
:- export chrc init(fdint).
:-        mode init(out) is det.
init(X), next_id(I) <=>
    X = fdint(I), next_id(I+1), bounds(X,-1000,1000).
init(X) <=> next_id(0), init(X).

%%% Stores the lower (L) and upper (U) bounds for a variable X.
%%% Two bounds/3 constraints are merged into a single constraint
%%% with the tightest bounds.
:- export chrc bounds(fdint,int,int).
:-        mode bounds(in,in,in) is semidet.
```

```
bounds(X,L,U) ==> U >= L.
bounds(X,L1,U1) \ bounds(X,L2,U2) <=>
         L1 >= L2, U2 >= U1 | true.
bounds(X,L1,U1), bounds(X,L2,U2) <=>
         bounds(X,max(L1,L2),min(U1,U2)).

%%% X = Y constraint.
:- chrc eq(fdint,fdint).
:- mode eq(in,in) is semidet.
eq(X,Y), bounds(X,LX,UX), bounds(Y,LY,UY) ==>
        bounds(Y,LX,UX),bounds(X,LY,UY).

%%% X >= Y constraint.
:- export chrc geq(fdint,fdint).
:-        mode geq(in,in) is semidet.
geq(X,Y), bounds(X,LX,UX), bounds(Y,LY,UY) ==>
        bounds(Y,LX,UY), bounds(X,LX,UY).

%%% X \= Y constraint.  Note that a symmetric copy of the neq
%%% constraint is added by the propagation rule, and the first
%%% rule ensures termination.
:- export chrc neq(fdint,fd_cint).
:-        mode neq(in,in) is semidet.
neq(X,Y) \ neq(X,Y) <=> true.
neq(X,Y) ==> neq(Y,X).
neq(X,Y),bounds(X,VX,VX),bounds(Y,VX,UY) ==> bounds(Y,VX+1,UY).
neq(X,Y),bounds(X,VX,VX),bounds(Y,LY,VX) ==> bounds(Y,LY,VX-1).

%%% X + Y = Z constraint.
:- chrc plus(fdint,fdint,fdint).
:- mode plus(in,in,in) is semidet.
plus(X,Y,Z),bounds(X,LX,UX),bounds(Y,LY,UY),bounds(Z,LZ,UZ) ==>
        bounds(X,LZ-UY,UZ-LY),bounds(Y,LZ-UX,UZ-LX),
        bounds(Z,LX+LY,UX+UY).
```

A.2 Ray Tracer in HAL CHR

The following is the full source code for a simple ray tracer implemented in HAL CHR. It supports spheres, planes and lightsources which can be placed anywhere in 3D space. It also supports shadows formed by objects blocking lightsources. The scene is read from standard input, for example the input

```
input_sphere(0.0,0.0,0.0,1.0,color(1.0,0.0,0.0)).
input_plane(0.0,0.0,1.0,0.0,color(0.0,0.0,1.0)).
input_light(0.0,0.0,5.0,color(1.0,1.0,1.0)).
input_eye(5.0).
```

creates a scene with one red sphere centered at $(0, 0, 0)$ with radius 1, a blue plane given by $(z = 0)$, and a white light source centered at $(0, 0, 5)$. The `input_eye(5.0)` sets the eye point (which is always on the z-axis) to $(0, 0, 5)$. The output is a 512×512 PPM image of the scene rendered from the eyepoint towards $(0, 0, 0)$. Note that PPM, or "Portable PixMap", is a very simple image file format.

```
:- module ray.

:- import int.
:- import float.
:- import math.
:- import chr.
:- import (io).
:- import term.
:- import term_io.
:- import delay.

:- export typedef color  -> color(float,float,float).
:-        typedef object ->
                   ( input_eye(float) ;
                     input_sphere(float,float,float,float,color) ;
                     input_plane(float,float,float,float,color) ;
                     input_light(float,float,float,color) ).
:-        typedef object_id = int.

%%% The main predicate of the program.   Reads the scene from stdin,
%%% then outputs the image to stdout.
:- export io pred main.
:-           mode main is det.
main :-
     io read_scene,
     io ppm_header,
     io render_pixels(0,0).

%%% Parses the input using the term and term_io standard libraries.
%%% Calls the appropriate CHR constraints based on the input.
:- io pred read_scene.
:-    mode read_scene is det.
read_scene :-
      io read_term(ReadTerm),
         ( ReadTerm = eof,
             true
         ; ReadTerm = error(_,_),
           io parse_error
         ; ReadTerm = term(_,Term),
             try_term_to_type(Term,Result),
```

```
            ( Result = ok(Object),
                ( Object = input_eye(Z),
                    eye(Z)
                ; Object = input_sphere(X,Y,Z,R,C),
                    sphere(X,Y,Z,R,C)
                ; Object = input_plane(A,B,C,D,Cl),
                    plane(A,B,C,D,Cl)
                ; Object = input_light(X,Y,Z,C),
                    light(X,Y,Z,C)
                ),
             io read_scene
            ; Result = error(Error),
               io parse_error(Error)
            )
          ).

:- io pred parse_error.
:-    mode parse_error is det.
parse_error :-
     io write_string("error while reading input!\n").

:- io pred parse_error(term_to_type_error(object)).
:-    mode parse_error(in) is det.
parse_error(mode_error(_,_)) :-
     io write_string("error while reading input! (mode error)\n").
parse_error(type_error(_,_,_,_)) :-
     io write_string("error while reading input! (type error)\n").

%%% Prints the standard PPM header (ascii-mode 512x512 255).
:- io pred ppm_header.
:-    mode ppm_header is det.
ppm_header :-
     io write_string("P3\n"),
     io write_string("512 512\n"),
     io write_string("255\n").

%%% Iterates through all the pixels of the image and renders them
%%% (by firing a ray for that pixel).  Outputs the resulting
%%% color to stdout (as part of the PPM image).
:- io pred render_pixels(int,int).
:-    mode render_pixels(in,in) is det.
render_pixels(X,Y) :-
        ( Y = 512 ->
            true
        ;   pixel(X,Y,C),
            C = color(R,G,B),
            floor_to_int(255.0*R,Ri),
```

```
            floor_to_int(255.0*G,Gi),
            floor_to_int(255.0*B,Bi),
          io write_int(Ri),
          io write_string(" "),
          io write_int(Gi),
          io write_string(" "),
          io write_int(Bi),
          io write_string(" "),
            NX = X + 1,
            ( NX = 512 ->
             io write_string("\n"),
             io render_pixels(0,Y+1)
            ;io render_pixels(NX,Y)
            )
        ).

%%% Auxiliary predicate calculates the intersection points between a
%%% ray and a sphere (if they exist).
:- pred sphere_intersection_calculation(float,float,float,float,float,
                        float,float,float,float,float,float,float).
:- mode sphere_intersection_calculation(in,in,in,in,in,in,in,in,in,in,
                        out,out) is semidet.
sphere_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,R,U1,U2) :-
        A = (X2-X1)*(X2-X1) + (Y2-Y1)*(Y2-Y1) + (Z2-Z1)*(Z2-Z1),
        B = 2.0*((X2-X1)*(X1-X3) + (Y2-Y1)*(Y1-Y3) + (Z2-Z1)*(Z1-Z3)),
        C = X3*X3 + Y3*Y3 + Z3*Z3 + X1*X1 + Y1*Y1 + Z1*Z1 -
                2.0*(X3*X1 + Y3*Y1 + Z3*Z1) - R*R,
        D = B*B - 4.0*A*C,
        D >= 0.0,
        U1 = (-B + sqrt(D))/(2.0*A),
        U2 = (-B - sqrt(D))/(2.0*A).

%%% Auxiliary predicate calculates the intersection point between a
%%% ray and a plane (if a single point exists).
:- pred plane_intersection_calculation(float,float,float,float,float,
                        float,float,float,float,float,float).
:- mode plane_intersection_calculation(in,in,in,in,in,in,in,in,in,in,
                        out) is semidet.
plane_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,A,B,C,D,U) :-
        UD = A*(X1-X2) + B*(Y1-Y2) + C*(Z1-Z2),
        UD \= 0.0,
        UN = A*X1 + B*Y1 + C*Z1 + D,
        U = UN/UD.

%%% Auxiliary predicate calculates the actual intersection point from
%%% the output (i.e. U) of the previous two predicates.
:- pred intersection_point(float,float,float,float,float,float,float,
```

```
                        float,float,float).
:- mode intersection_point(in,in,in,in,in,in,in,out,out,out) is det.
intersection_point(X1,Y1,Z1,X2,Y2,Z2,U,PX,PY,PZ) :-
        PX = X1 + U*(X2-X1),
        PY = Y1 + U*(Y2-Y1),
        PZ = Z1 + U*(Z2-Z1).

%%% Auxiliary predicate calculates the distance between two points.
:- pred distance(float,float,float,float,float,float,float).
:- mode distance(in,in,in,in,in,in,out) is det.
distance(X1,Y1,Z1,X2,Y2,Z2,D) :-
        DX = X1-X2,
        DY = Y1-Y2,
        DZ = Z1-Z2,
        D = sqrt(DX*DX + DY*DY + DZ*DZ).

%%% Auxiliary predicate converts a ray into a normalised vector.
:- pred normalized_vector(float,float,float,float,float,float,float,
                        float,float).
:- mode normalized_vector(in,in,in,in,in,in,out,out,out) is det.
normalized_vector(X1,Y1,Z1,X2,Y2,Z2,VX,VY,VZ) :-
        distance(X1,Y1,Z1,X2,Y2,Z2,L),
        IL = 1.0/L,
        VX = IL*(X2-X1),
        VY = IL*(Y2-Y1),
        VZ = IL*(Z2-Z1).

%%% Auxiliary predicate calculates a plane's normal.
:- pred normalized_plane_vector(float,float,float,float,float,float,
                        float).
:- mode normalized_plane_vector(in,in,in,in,out,out,out) is det.
normalized_plane_vector(A,B,C,D,VX,VY,VZ) :-
        VX0 = A+D,
        VY0 = B+D,
        VZ0 = C+D,
        L = sqrt(VX0*VX0 + VY0*VY0 + VZ0*VZ0),
        IL = 1.0/L,
        VX = IL*VX0,
        VY = IL*VY0,
        VZ = IL*VZ0.

%%% Auxiliary predicate calculates the dot product of two vectors.
:- pred dot_product(float,float,float,float,float,float,float).
:- mode dot_product(in,in,in,in,in,in,out) is det.
dot_product(X1,Y1,Z1,X2,Y2,Z2,D) :-
        D = X1*X2 + Y1*Y2 + Z1*Z2.
```

```
%%% Auxiliary predicate tests if a sphere blocks a ray.
:- pred sphere_blocks(float,float).
:- mode sphere_blocks(in,in) is semidet.
sphere_blocks(U1,U2) :-
        ( U1 > 0.0, U1 < 1.0 ->
            true
        ; U2 > 0.0, U2 < 1.0 ->
            true
        ;   fail).

%%% Auxiliary predicate tests if a plane blocks a ray.
:- pred plane_blocks(float).
:- mode plane_blocks(in) is semidet.
plane_blocks(U) :-
        U > 0.0,
        U < 1.0.

%%% Auxiliary predicate blends two colors.
:- pred blend_colors(float,color,color,color).
:- mode blend_colors(in,in,in,out) is det.
blend_colors(D,C1,C2,C3) :-
        C1 = color(R1,G1,B1),
        C2 = color(R2,G2,B2),
        R3 = D*R1*R2,
        G3 = D*G1*G2,
        B3 = D*B1*B2,
        C3 = color(R3,G3,B3).

%%% Constraint representing a sphere.
:- chrc sphere(float,float,float,float,color).
:- mode sphere(in,in,in,in,in) is det.

%%% Constraint representing a sphere & allocated an object_id.
:- chrc sphere(object_id,float,float,float,float,color).
:- mode sphere(in,in,in,in,in,in) is det.

%%% Constraint representing a plane.
:- chrc plane(float,float,float,float,color).
:- mode plane(in,in,in,in,in) is det.

%%% Constraint representing a sphere & allocated an object_id.
:- chrc plane(object_id,float,float,float,float,color).
:- mode plane(in,in,in,in,in,in) is det.

%%% Auxiliary constraint to store the next free object_id.
:- chrc object_id(object_id).
:- mode object_id(in) is det.
```

```
%%% The following rules allocate object_id for spheres and planes.
object_id(_) \ object_id(_) <=>
        true.
object_id(I), sphere(X,Y,Z,R,C) <=>
        sphere(I,X,Y,Z,R,C),
        object_id(I+1).
sphere(X,Y,Z,R,C) <=>
        sphere(0,X,Y,Z,R,C),
        object_id(1).
object_id(I), plane(A,B,C,D,Cl) <=>
        plane(I,A,B,C,D,Cl),
        object_id(I+1).
plane(A,B,C,D,Cl) <=>
        plane(0,A,B,C,D,Cl),
        object_id(1).

%%% Enforce functional dependencies, i.e. the object_id is
%%% unique to each constraint.  These rules are mainly for the
%%% compiler's benefit.
sphere(I,_,_,_,_,_) \ sphere(I,_,_,_,_,_) <=>
        true.
plane(I,_,_,_,_,_) \ plane(I,_,_,_,_,_) <=>
        true.

%%% Constraint to store the eyepoint.
:- chrc eye(float).
:- mode eye(in) is det.

%%% There is only one eyepoint.
eye(_) \ eye(_) <=>
        true.

%%% A call to pixel(X,Y,C) calculates the color C for pixel
%%% (X,Y).
:- chrc pixel(int,int,color).
:- mode pixel(in,in,out) is det.

%%% Constraint representing a lightsource.
:- chrc light(float,float,float,color).
:- mode light(in,in,in,in) is det.

%%% Constraint representing the ray fired from the eyepoint
%%% to some point in space.
:- chrc ray(float,float,float,float,float,float).
:- mode ray(in,in,in,in,in,in) is det.
```

```
%%% Constraint representinf a ray fired from a lightsource
%%% to an intersection point.  This is used to calculate
%%% shdows.
:- chrc light_ray(float,float,float,color).
:- mode light_ray(in,in,in,in) is det.

%%% Create the ray.
pixel(X,Y,_), eye(Z) ==>
        to_float(X,Xf),
        to_float(Y,Yf),
        ray(0.0,0.0,Z,0.5 - Xf/512.0,0.5 - Yf/512.0,Z - 1.0).

%%% Create the light rays.
pixel(X,Y,_), light(LX,LY,LZ,C) ==>
        light_ray(LX,LY,LZ,C).

%%% Get the color and clean up.
pixel(_,_,C) <=>
        clean_up,
        get_color(C).

%%% Represents an intersection from the eye ray and an object.
%%% Note that we are only interested in the "closest"
%%% intersection.
:- chrc intersection(float,float,float,float,object_id,color).
:- mode intersection(in,in,in,in,in,in) is det.
intersection(_,_,_,_,_,_) \ intersection(_,_,_,_,_,_) <=>
        true.
intersection(_,_,_,L1,_,_) \ intersection(_,_,_,L2,_,_) <=>
    L1 =< L2 |
        true.

%%% The following rules calculate what the eye ray intersects.
ray(X1,Y1,Z1,X2,Y2,Z2), sphere(I,X3,Y3,Z3,R,C) ==>
        ( sphere_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,
                    R,U1,U2) ->
            ( U1 > 0.0 ->
                intersection_point(X1,Y1,Z1,X2,Y2,Z2,U1,PX1,PY1,PZ1),
                distance(X1,Y1,Z1,PX1,PY1,PZ1,L1),
                intersection(PX1,PY1,PZ1,L1,I,C)
            ;   true),
            ( U2 > 0.0 ->
                intersection_point(X1,Y1,Z1,X2,Y2,Z2,U2,PX2,PY2,PZ2),
                distance(X1,Y1,Z1,PX2,PY2,PZ2,L2),
                intersection(PX2,PY2,PZ2,L2,I,C)
            ;   true
            )
```

```
        ;   true
        ).
ray(X1,Y1,Z1,X2,Y2,Z2), plane(I,A,B,C,D,Cl) ==>
        ( plane_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,A,B,C,D,U),
          U > 0.0 ->
            intersection_point(X1,Y1,Z1,X2,Y2,Z2,U,PX,PY,PZ),
            distance(X1,Y1,Z1,PX,PY,PZ,L),
            intersection(PX,PY,PZ,L,I,Cl)
        ; true
        ).
ray(_,_,_,_,_,_) <=>
        true.

%%% The current color being accumulated for the current pixel.
:- chrc color(color).
:- mode color(in) is det.

%%% Adds a color to the current pixel.
:- chrc add_color(color).
:- mode add_color(in) is det.

%%% Gets the value of the color for the current pixel.
:- chrc get_color(color).
:- mode get_color(out) is det.

%%% Only one current color.
color(_) \ color(_) <=>
        true.

add_color(C1), color(C2) <=>
        C1 = color(R1,G1,B1),
        C2 = color(R2,G2,B2),
        R3 = R2 + R1,
        G3 = G2 + G1,
        B3 = B2 + B1,
        C3 = color(R3,G3,B3),
        color(C3).
add_color(C) <=>
        color(C).

color(C1), get_color(C2) <=>
        C1 = color(R1,G1,B1),
        ( R1 > 1.0 ->
            R2 = 1.0
        ;   R2 = R1),
        ( G1 > 1.0 ->
            G2 = 1.0
```

```
        ;   G2 = G1),
        ( B1 > 1.0 ->
            B2 = 1.0
        ;   B2 = B1),
        C2 = color(R2,G2,B2).
get_color(C) <=>
        C = color(0.0,0.0,0.0).

%%% Constraint representing a light ray between two points in space.
:- chrc light_ray(float,float,float,float,float,float,color,object_id).
:- mode light_ray(in,in,in,in,in,in,in,in) is det.

%%% Create light rays from light sources and intersection points.
intersection(X,Y,Z,_,I,_) \ light_ray(LX,LY,LZ,C) <=>
        light_ray(LX,LY,LZ,X,Y,Z,C,I).
light_ray(_,_,_,_) <=>
        true.

%%% Calculate shadows, i.e. remove light rays that are blocked by
%%% an object.
sphere(I,X3,Y3,Z3,R,_) \ light_ray(X1,Y1,Z1,X2,Y2,Z2,_,J) <=>
    I \= J,
    sphere_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,
                Z3,R,U1,U2),
    sphere_blocks(U1,U2) |
        true.
plane(I,A,B,C,D,_) \ light_ray(X1,Y1,Z1,X2,Y2,Z2,_,J) <=>
    I \= J,
    plane_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,A,B,C,D,U),
    plane_blocks(U) |
        true.

%%% Calculate the effect of the surviving light rays.
sphere(I,X3,Y3,Z3,_,C1) \ light_ray(X1,Y1,Z1,X2,Y2,Z2,C2,I) <=>
        normalized_vector(X3,Y3,Z3,X2,Y2,Z2,VX1,VY1,VZ1),
        normalized_vector(X2,Y2,Z2,X1,Y1,Z1,VX2,VY2,VZ2),
        dot_product(VX1,VY1,VZ1,VX2,VY2,VZ2,D),
        ( D =< 0.0 ->
            true
        ;   blend_colors(D,C1,C2,C3),
            add_color(C3)
        ).
plane(I,A,B,C,D,C1) \ light_ray(X1,Y1,Z1,X2,Y2,Z2,C2,I) <=>
        normalized_plane_vector(A,B,C,D,VX1,VY1,VZ1),
        normalized_vector(X2,Y2,Z2,X1,Y1,Z1,VX2,VY2,VZ2),
        dot_product(VX1,VY1,VZ1,VX2,VY2,VZ2,D0),
        abs(D0,D1),
```

```
        blend_colors(D1,C1,C2,C3),
        add_color(C3).
light_ray(_,_,_,_,_,_,_,_) <=>
        true.

%%% Auxiliary constraint that resets the store for the next pixel.
:- chrc clean_up.
:- mode clean_up is det.
clean_up \ intersection(_,_,_,_,_,_) <=>
        true.
clean_up <=>
        true.
```