Execution models for Constraint Programming: kernel language design through semantics equivalence.

Thierry Martinez

UNIVERSITÉ SORBONNE
PARIS CITÉ
Université Paris Diderot

ÉCOLE DOCTORALE : SCIENCES MATHÉMATIQUES DE PARIS CENTRE

Équipe–projet Lifeware, Inria

DOCTORAT
Informatique

Thierry Martinez

Execution models for Constraint Programming: kernel language design through semantics equivalence.

Modèles d'exécution pour la programmation par contraintes :
conception d'un langage noyau par le biais d'équivalences sémantiques

Thèse dirigée par François Fages

Soutenue le 17 décembre 2015

JURY

M. Roberto Di Cosmo	Professeur des universités Université Paris 7	Président du jury
M. François Fages	Directeur de recherche Inria	Directeur de thèse
M. Tom Schrijvers	Professeur Katholieke Universiteit Leuven	Rapporteur
M. Maurizio Gabbrielli	Professeur Universita' di Bologna	Rapporteur
M. Jean Goubault-Larrecq	Professeur École normale supérieure de Cachan	Examinateur
M. Franck Valencia	Chargé de recherche CNRS	Examinateur

Contents

1 Introduction **9**
- 1.1 Committed-Choice Programming . . . 10
- 1.2 Constraint Model and Search . . . 11
- 1.3 Angelic Programming . . . 12
- 1.4 Thesis . . . 14

2 Ask-lifting: from LLCC to CHR **17**
- 2.1 Syntax & Semantics of CHR and LLCC . . . 17
  - 2.1.1 Syntax and Semantics of CHR . . . 18
  - 2.1.2 Syntax and Semantics of LLCC . . . 21
  - 2.1.3 Circumscribing non-determinism in CHR and LLCC operational semantics . . . 24
- 2.2 Translations between sub-languages CSR and flat-LLCC . . . 25
  - 2.2.1 From Constraint Simplification Rules (CSR) to flat-LLCC . 26
  - 2.2.2 From flat-LLCC to CSR . . . 27
  - 2.2.3 CHR Linear-Logic and Phase Semantics Revisited . . . 29
- 2.3 Ask-lifting: encoding LLCC into CSR . . . 30
  - 2.3.1 Preliminary step: labeling LLCC-agents . . . 30
  - 2.3.2 The ask-lifting transformation . . . 32
- 2.4 Encoding the call-by-value λ-calculus in CHR . . . 33

3 Reified Search: search strategies as constraint solving **35**
- 3.1 Constraint Satisfaction . . . 35
  - 3.1.1 Constraints as Relational Languages . . . 35
  - 3.1.2 Propagators and Consistency . . . 39
  - 3.1.3 Abstractions . . . 41
- 3.2 The Language ClpZinc . . . 42
- 3.3 Extending Zinc with CLP Clauses . . . 45
  - 3.3.1 Partial Evaluation of ClpZinc into And/Or Trees . . . 45
  - 3.3.2 Compiling And/Or Trees into Zinc Reified Constraints . . . 46
- 3.4 Computation Results on Korf's Square Packing Benchmark . . . 49

3.5 LDS and SBDS as Strategy Transformers in ClpZinc 53
3.5.1 Limited Discrepancy Search 53
3.5.2 Symmetry Breaking During Search 54
3.6 Beyond Tree Search Strategies 55
3.7 ClpZinc Models of First-Fail and Middle-out Strategies 56
3.8 ClpZinc Model of Symmetry Breaking During Search 58

4 Angelic Programming: from compound guards to atomic kernel for LLCC **63**
4.1 Angelic Semantics and Derivation nets 63
4.1.1 Process calculi 63
4.1.2 Nets for sets of reductions 64
4.1.3 Derivation nets for Constraint Simplification Rules (CSR) . 66
4.1.4 Sharing strategies 69
4.1.5 Derivation nets in presence of Propagation Rules 70
4.2 Angelic Programming 71
4.2.1 Head Decomposability 71
4.2.2 Controlling the Angelism 71
4.3 Applications 72
4.3.1 Angelic CHR$\vee$ 72
4.3.2 Meta-interpreters 72
4.3.3 User-defined Indexation 73
4.3.4 CHR with Ask and Tell 73
4.3.5 Hierarchical Definition of a Rational Terms Constraint Solver 84
4.4 The SiLCC Programming Language 88
4.4.1 Procedures 89
4.4.2 Sequencing 90
4.4.3 Arithmetic and functional notation 92
4.4.4 Conditional 92
4.4.5 Angelism 94
4.4.6 Modules and the package system 95
4.4.7 Recursion 96
4.4.8 Lists and iterations 97
4.4.9 Pattern-matching and terms 97
4.4.10 References 99
4.4.11 Records 100
4.4.12 Hash-tables and argument indexing 100
4.4.13 Asks on variable number of tokens 101

5 Conclusion **105**
5.1 Committed-Choice Semantics for LLCC through CHR 106
5.2 Constraint Model and Search 106
5.3 Angelic Programming 107

Summary

Logic programming and constraint programming are two declarative programming paradigms which rely on the identification of programs to theories, and programming to modeling. Execution models result from the operational interpretation of logical provability in logic programming, and of constraint propagation in constraint programming. However, the control of execution is crucial for the practicability of these schemes and extra-logical traits are thus added in those programming systems, with the classical slogans "logic program = logical theory + control", "constraint program = constraint model + search".

This thesis investigates execution models in which control and search can be shifted into the logic or the constraint model, while preserving the semantics. The three parts of the thesis correspond to the three semantics equivalence that are showed: the first between two committed-choice forward-chaining logic languages, the second between constraint logic programs and constraint models, and the third between guard semantics in angelic settings. Each of these equivalence is constructive in the sense that there exists an encoding that enables the compilation from one of the paradigm to the other.

First, we show that simple program transformations exist back and forth between Constraint Handling Rules (CHR) and Linear Logic Concurrent Constraint (LLCC) languages, making them semantically equivalent even if syntactically different, which closes the question of implementing a committed-choice semantics for LLCC by using CHR as kernel language.

Secondly, we show that a wide variety of search procedures can be internalized in the constraint model with a fixed enumeration strategy. Transforming search procedures into constraint satisfaction problems presents several advantages: (1) it makes search strategies declarative and modeled as constraint satisfaction problems; (2) it makes it possible to express search strategies in existing front-end modeling languages without any extension; (3) it opens up constraint propagation algorithms to search constraints and to the implementation of novel search procedures based on constraint propagation. This is illustrated with the design of the ClpZinc modeling language, with an angelic interpretation of Horn clauses which allows compilation to a reification-based constraint kernel.

Finally, in concurrent constraint logic programming, committed-choice semantics create a hierarchy of non-equivalent semantics axed on the expressive power of synchronization mechanism. We show that the hierarchy of guard semantics collapses with angelic semantics, allowing the most primitive synchronization mechanism to encode all the others. The main consequence of this collapse is the identification of a kernel language, with a primitive synchronization mechanism and an elementary constraint system which are sufficient to reconstruct the other forms of synchronizations and other constraint systems as modules, in the software engineering sense.

Chapter 1

Introduction

Program semantics are the mathematical abstractions of program executions. In the settings of logic programming [16, 50], the program semantics results from the operational interpretation of logical provability: the execution is a proof search procedure for the program interpreted as a logical property expressed in a computational fragment of a logic [56]. In the settings of constraint programming, derived from constraint propagation algorithms first introduced by Waltz for computer vision [79], the execution is an exploration procedure of a combinatorial space searching for a variable assignment satisfying all the relations, the "constraints". In the settings of concurrent programming, the execution is a certain interleaving of computation paths and message transfers. In all these settings, the execution path does not directly follow the program structure: the precise execution relies on the proof method, the search strategy or the scheduler. The precise semantics of the programs is therefore intimately linked with the execution model.

Logic programming and constraint programming have been linked into constraint logic programming [46] which generalizes from the Herbrand domain of terms with equalities to arbitrary domains of partially-known values with constraints. In the constraint programming settings, constraint propagators act concurrently to reduce variable domains. The use of variable domains as communication channel inspired the foundation of the class CC of *Concurrent Constraint* programming languages [54, 65] which rely on a model of concurrent computation, where agents communicate through a shared constraint store, with a synchronization mechanism based on constraint entailment. In classical constraint settings, the store evolves monotonically, similarly to the built-in constraint store of CHR. The LLCC languages [64, 28] introduce linear constraint systems, based on Girard's intuitionistic linear logic (ILL) [34]. A remarkable kind of linear constraints are linear tokens [28], which can be freely added or consumed, comparably to CHR constraints. Linear logic leads to a natural semantics for classical CC languages as well [28].

Logic programming and constraint programming share the common trait of lifting programming into modeling: as in linear programming [14], or mathematical programming more generally, the paradigm tends to identify the program with a model and the execution with its resolution. However, since the introduction of Prolog [16, 50], logic programming has been living with the dichotomy: "programs = logic + control". Similarly, constraint programming is traditionally presented as the combination of two components: a constraint model and a search procedure [78]. In the general case, concurrent programs are not necessary confluent and their execution relies upon scheduler choices. While denotational semantics are faithful with respect with the logic of a logical program, the model of a constraint satisfaction problem or the set of all execution paths of a concurrent program, the control suffers from being only described by a procedural interpretation.

This thesis investigates how execution models can internalize the control into the logic.

1.1 Committed-Choice Programming

Constraint Handling Rules (CHR) [30] is a rule-based declarative programming language. CHR and LLCC have been developped independently and with distinct concerns. More recently, a precise declarative semantics for CHR has been described in linear logic [12]. Implementations of CHR follow a committed-choice forward-chaining execution model: the non-determinism of the abstract semantics is partly refined with extra-logical syntactic convention on the program order and possibly notations for weighted semantics (with priorities or probabilities), and partly left unspecified in the underlying compiler.

In CHR, programs are sets of transformation rules on constraint stores. Some constraints are built-ins and can only be accumulated into the store. Other constraints are user-defined and can be added or deleted. Although initial motivations were the definition of constraint solvers and propagators, nowadays applications include typing [18, 76], software testing [59], scheduling [4] and so on.

The linear-logic CC language (or LLCC) provides the non-monotonous traits for imperative programming in addition to traits for logic, concurrent and constraint programming, the whole in a simple and pure semantics. This semantics identifies a fragment of first-order linear logic as a monoparadigm programming language with a rich expressive power.

Two translations from CHR to LLCC and back are proposed, both preserving the semantics. Strong bisimilarity results are formulated. As direct corollary, we obtain a natural encoding of the λ-calculus in CHR. While existence of low-level translations is guaranteed by Turing-completeness *via* a compilation process, there are more fine-grained criteria to compare expressiveness [36]. In particular,

translations presented here are natural and (relatively) agnostic with respect to the constraint theory.

As every logic-based language, the declarativity of Constraint Handling Rules [31, 75] relies on the logical interpretation of the programs. However, this interpretation hides syntactic conventions, like the order of the rules, distinguished abbreviations such as propagation rules, and annotations which control the effective execution of the program. These control features are formally described in a hierarchy of semantics, from the abstract semantics ω_{va} [31] to more fine-grained semantics, describing the handling of propagations (the theoretical semantics ω_{t}), of the rule ordering (the refined semantics ω_{r} [26]), or of annotations like priorities [22] or probabilities [72].

This thesis formalizes connections between CHR with naive operational semantics and LLCC.

1.2 Constraint Model and Search

Front-end modeling languages are designed for solving problems independently of the solvers, and solving them with generic solvers using fixed search procedures (e.g. Essence [29]), or contain special features for specifying the search strategy for the constraint solvers (e.g. Zinc [57]).

In this chapter, we show that a completely different approach for specifying search is possible, by internalizing the search procedure in the constraint model with a fixed enumeration strategy. In principle, transforming search procedures into constraint satisfaction problems presents several advantages:

1. it makes search strategies declarative, and modeled as constraint satisfaction problems;

2. it makes it possible to express search strategies in existing front-end modeling languages without any extension;

3. it opens up constraint propagation algorithms to search constraints and to the implementation of novel search procedures based on constraint propagation.

The idea of this transformation is to associate to each choice point a reified constraint with an auxiliary model variable for representing that choice (e.g. value enumeration, domain splitting or any constraint). The search heuristic can then be specified simply by the enumeration strategy for the choice variables. This approach is not limited to static search procedures in which all choice points are precisely known statically, but can accommodate dynamic search strategies, such

as dichotomic or interval splitting search [71] for example. In constraint programming, dynamic search procedures rely on the values of indexicals (domain size, minimum value, etc.). They are expressed in the framework presented here by extending the enumeration strategy with annotations that assign the values of indexicals to auxiliary model variables. Static search procedures do not rely on the values of indexicals and their encoding do not need any specific support on the solver-side. The encoding of dynamic search procedures can be run through simple additions in the solvers for providing the capability to query the values of indexicals.

In this chapter, to make concrete the presentation of the transformation, we consider the Zinc modeling language and introduce ClpZinc [1], a language extending Zinc with the ability to describe new relations by Horn clauses. The choice of CLP as a specification language for search procedures is guided by CLP being the smallest language with the addition of constraint to the store as primitive and closed by conjunction and disjunction (for expressing choices), and with a general form of recursion. Given a constraint system $\mathcal{X}$ (e.g. finite domains) and the Herbrand constraint system $\mathcal{H}$, we describe a partial evaluation procedure to transform any terminating CLP($\mathcal{X} + \mathcal{H}$) goal to an and/or tree with constraints over $\mathcal{X}$.

1.3 Angelic Programming

Finally, this thesis proposes an alternative execution model which explores all the possible choices, by opposition to the committed-choice strategy. This execution model is angelic in the sense that if there exists a successful execution strategy (with respect to a given observable), then this strategy will be found. Formally, the set of computed goals is complete with respect to the set of the logical consequences of the interpretation of the initial goal in linear logic. In practice, this chapter introduces a new data representation for sets of goals, the derivation nets. Sharing strategies between computation paths can be defined for derivation nets to make execution algorithmically tracktable in some cases where a naive exploration would be exponential. Control for refined execution is recovered with the introduction of user constraints to encode sequencing, fully captured in the linear-logic interpretation. As a consequence of angelic execution, CHR rules become decomposable while preserving accessibility properties. This decomposability makes natural the definition in angelic CHR of meta-interpreters to change the execution strategy. More generally, arbitrary computation can be interleaved during head matching,

[1] The Clp2Zinc compiler that transforms ClpZinc models into MiniZinc is available for download, together with patches for Choco, JaCoP, SICStus, Gecode and or-tools: `http://lifeware.inria.fr/~tmartine/clp2zinc/`

for custom user constraint indexation and deep guard definition.

CHR enjoys two logical interpretation: the first to have been introduced historically interprets rules and goals as first-order classical logic formulae; more recently [12], an interpretation as first-order linear logic [34] formulae has been given. The latter provides a finer reading of the dynamics of the rules and will be the logical interpretation considered in this chapter.

All these semantics are correct with respect to the linear-logic interpretation: if a configuration is reachable through any of these operational semantics, then this configuration is indeed a linear-logic consequence of the initial goal. However, only the abstract semantics enjoys completeness: the purpose of all other semantics is to provide syntactic construction to force the execution to choose some particular branches. The downside is that these scheduling choices escape the declarative framework provided by logic. The programmer should ensure that the scheduler can only make the good choice, either by writing a confluent program or by relying on extra-logical traits (order of the rules, priorities, etc.) to drive the scheduler.

Focusing on completeness entails the exploration of all the logical consequences of the interpretation of the initial goal in linear logic. For this purpose, we propose angelic scheduling as an alternative execution model for CHR. Observationaly, the scheduler always makes the good choice: more precisely, if a successful (i.e., non-blocking) choice exists, it will be explored. Accessible configurations exactly match the set of logical consequences of the linear interpretation of the initial goal: the operational behavior is fully described by the linear-logic interpretation, including the control. More formally, linear logic is the most faithful logic for CHR [12], since it captures the non-monotonous evolution of configurations. Control structures like sequencing and branching have natural encoding in this logic and their usage for CHR have already been showed through the log-linear encoding of RAM machines [74].

In angelic settings, the atomicity of head consumption is not essential, in opposition to the committed-choice case. Since absence of user constraints cannot be observed, partial head consumptions just lead to silent unsuccessful computation branches. This property allows the interleaving of arbitrary computations between multiple head consumptions. Meta-interpreters for CHR rules can therefore be written by sequencing the consumption of the successive parts of the head. Specific representations can be chosen for some heads to enable user-defined indexation strategy. To reduce the combinatorial explosion among computation branches, the formalism of derivation nets is introduced: this formalism provides a graphical representation for sets of computation paths. Non-determinism during the execution of a CHR program can be intrinsic to the rule dynamics, and all choices should be explored, but the abstract operational semantics suffers from a large part of scheduling non-determinism between independent paths of the com-

putation that should be quotiented for a tractable execution. The derivation nets are a convenient representation to define sharing strategies between computation paths to eliminate scheduling non-determinism. Two decidable sharing strategies are explored in this chapter. The first strategy shows that optimal sharing is decidable but is computationaly expensive. The second one is polynomial in the worst case and induces essentially a constant overhead in practice while being optimal relatively to a conservative interpretation of user constraint identity.

Angelic semantics have been identified as the natural semantics for Concurrent Constraint (CC) programming languages [47] since the very beginning of the introduction of this language family: in this forward-chaining framework, the set of accessible computations is more natural to link with a logical interpretation than a particular computation path. However, the CC language and its angelic semantics is considered in [47] as an abstract language to reason about concurrency-related questions that can be captured in this formalism: there is no consideration about implementation. Moreover whenever CC languages have only a monotonous interpretation in classical logic, LLCC and CHR handle non-monotonous traits with consumptions.

Besides these algorithmic results, the angelic execution of programs has numerous semantically good properties. The behavior of programs is by construction precisely captured by the logical interpretation. Committed-choice semantics create a hierarchy of non-equivalent semantics axed on the expressive power of synchronization mechanism: this hierarchy collapses with angelic semantics, allowing the most primitive synchronization mechanism to encode all the others. The main consequence of this collapse is the identification of a kernel language, with this primitive synchronization mechanism and an elementary constraint system, sufficient to reconstruct the other forms of synchronizations and other constraint systems as modules, in the software engineering sense. The kernel language can be seen has a generalization of Warren's abstract machine for Prolog [80].

1.4 Thesis

This thesis proposes three program transformations that describe three execution models for constraint programming that are guided by lifting programming into modeling. Constraint Handling Rules are used as a compilation target for the committed-choice semantics for LLCC, constraint satisfaction problems are used as a compilation target for search strategies, and logical interpretation of agents, omniscient over execution paths, enables extra-logical control operators to be modeled as blocked paths.

- The first chapter, "Ask-lifting: from LLCC to CHR", shows that the question of implementing a committed-choice semantics for LLCC can be reduced to a

program transformation to the actively developed field of CHR compilation,

- The second chapter, "Reified Search: search strategies as constraint solving", shows that the fundamental dichotomy between modeling and search in constraint programming can in fact be eliminated by showing how search strategies can be modeled as constraint satisfaction problems with fixed enumeration strategies,

- The third chapter, "Angelic Semantics: from compound guards to atomic kernel for LLCC", shows that the question of choosing sophisticated semantics for control under the presence of non-elementary asks can be reduced to a simple kernel language with atomic guards.

Through these three semantics equivalences, between LLCC and CHR, between tree-search exploration and constraint satisfaction, and between deep guards and atomic guards in angelic settings, this thesis propose three novel compilation schemes, with CHR, CSP with fixed enumeration strategy, and atomic asks as kernel languages.

Chapter 2

Ask-lifting: from LLCC to CHR

Section 2.1 presents CHR and LLCC in full generality and recalls some already published and well-known results. Section 2.2 focuses on distinguished subsets *Constraint Simplification Rules* (CSR) and flat-LLCC, provides translations between these two subsets. Linear logic semantics [12] and phase semantics [39] of CHR are recovered as corollary. Section 2.3 introduces the *ask-lifting* transformation from full LLCC to flat-LLCC. Section 2.4 presents the encoding of the call-by-value λ-calculus in CHR.

Related work

The adaptations of functional concepts in LLCC languages have been initiated with the embedding of closures and modules, leading to an encoding of λ-calculus in LLCC [41]. This chapter pursues the effort of transposing results in functional languages to concurrent constraint systems.

The translation from full LLCC to CHR relies on *ask-lifting*. This is a transformation comparable to the λ-lifting [48] for functional languages: the common idea is the materialization of the environment in data structures, *i.e.* values in functional languages or tokens in LLCC.

Flattening nested programming structures to CHR programs was suggested in [6] for connecting the Celf system [66] to CHR but, to our knowledge, no formal description of the transformation has been published.

2.1 Syntax & Semantics of CHR and LLCC

Let $\mathcal{V}$ be a set of variables, and Σ a signature for constant, function and predicate symbols. The set of free variables of a formula e is denoted $\mathrm{fv}(e)$, a sequence of variables is denoted by $\boldsymbol{x}$. $e[\boldsymbol{t}/\boldsymbol{x}]$ denotes the formula e in which free occurrences of

variables $\boldsymbol{x}$ are substituted by terms $\boldsymbol{t}$ (with the usual renaming of bound variables to avoid variable clashes).

For a set S, $S^\star$ denotes the set of finite sequences of elements of S and $\mathcal{M}(S)$ denotes the set of finite multi-sets of elements of S. More formally, $(S^\star; \cdot; \varepsilon)$ denotes the free monoid and $(\mathcal{M}(S); , ; \varnothing)$ the free commutative monoid over S. For relations $\mathcal{R}$ and $\mathcal{R}'$, $a\,\mathcal{R}\cdot\mathcal{R}'\,c$ if there exists b such that $a\,\mathcal{R}\,b\,\mathcal{R}'\,c$. For a relation $\rightarrow$, $\xrightarrow{\star}$ is the reflexive and transitive closure of $\rightarrow$.

2.1.1 Syntax and Semantics of CHR

Syntax.

Let $\mathcal{P}_b$ and $\mathcal{P}_c$ be two disjoint subsets of predicate symbols in Σ. Predicates built from Σ with predicate symbols in $\mathcal{P}_b$ are *atomic built-in constraints*, their set is denoted $\mathcal{B}_0$. *Built-in constraints* are conjunctions of atomic built-in constraints, their set is denoted $\mathcal{B}$. Predicates built from Σ with predicate symbols in $\mathcal{P}_c$ are *atomic* CHR *constraints*, their set is denoted $\mathcal{U}_0$. CHR *constraints* are (finite) multi-sets of atomic CHR constraints, their set is denoted $\mathcal{U}$. A *goal* is a multi-set of built-in constraints and CHR constraints.

Built-in constraints are supposed to include the syntactic equality $=$. There is a *constraint theory* CT over the built-in constraints: CT is supposed to be a non-empty, consistent and decidable first-order theory. For two multi-sets $H = (H_1, \ldots, H_m)$ and $H' = (H'_1, \ldots, H'_n)$, $H \doteq H'$ denotes the formula $H_1 = H'_1 \wedge \cdots \wedge H_n = H'_n$ if $m = n$, and **false** if $m \neq n$ [3].

Definition 1 (Syntax)**.** A CHR *program* is a sequence of rewriting rules, or CHR *rules*, each rule being denoted $\langle H \backslash H' \Leftrightarrow G \mid B\rangle$ where *heads* H and H' are CHR constraints such that $\langle H, H'\rangle \neq \varnothing$, the *guard* G is a built-in constraint, and the *body* B is a goal.

A *simplification rule* is a rule where H' is empty, and is denoted $H' \Leftrightarrow G \,|\, B$. A *propagation rule* is a rule where H' is empty, denoted $H \Rightarrow G \mid B$. A *simpagation rule* is a rule where H and H' are both non-empty. H and H' cannot be both empty.

Example 1. *The* CHR *program below, adapted from [10], describes the* dining philosophers *protocol [24], where N philosophers are sitting around a table and alternate thinking and eating. N forks are dispatched between them. Each philosopher is in competition with her neighbors to take her two adjacent forks and eat.*

```
diner(N) ⇔  recphilo (0,  N).
recphilo (I,  N) ⇔
    J is  (I + 1) mod N, philo(I,  J),  fork(I),
```

$$
\begin{array}{l}
\quad \mathit{nextphilo}(I,\ N).\\
\mathit{nextphilo}(I,\ N) \Leftrightarrow\ I < N - 1 \mid\\
\quad J\ \mathbf{is}\ I + 1,\ \mathit{recphilo}(J,\ N).\\
\mathit{philo}(I,\ J) \setminus \mathit{fork}(I),\ \mathit{fork}(J) \Leftrightarrow\ \mathit{eat}(I,\ J).\\
\mathit{eat}(I,\ J) \Leftrightarrow\ \mathit{fork}(I),\ \mathit{fork}(J).
\end{array}
$$

Logical Semantics.

CHR programs enjoy two declarative semantics: one in classical logic for left-linear programs [30], and one in linear logic without restriction [12]. A rule is *left-linear* if a user defined constraint matches at most one constraint in its head. A program is left-linear when all rules are.

In classical logic, a multiset M of constraints is interpreted by the conjunction $M^\dagger = \langle \bigwedge_{c \in \mathrm{sup}(M)} c \rangle$ of its elements. For each CHR rule:

$$\left(H \backslash H' \Rightarrow G \;\middle|\; B_\mathcal{X}, B_u\right)^\dagger = \forall \boldsymbol{x}(\exists \boldsymbol{y}(G) \wedge H^\dagger \rightarrow (H'^\dagger \leftrightarrow \exists \boldsymbol{z}(G \wedge B_\mathcal{X} \wedge B_u^\dagger)))$$

where $\boldsymbol{x} = \mathrm{fv}(H_0, H_1)$, $\boldsymbol{y} = \mathrm{fv}(G) \setminus \mathrm{fv}(H_0, H_1)$ and $\boldsymbol{z} = \mathrm{fv}(G, B_\mathcal{X}, B_u) \setminus \mathrm{fv}(H_0, H_1)$. The logical interpretation of a program $(P)^\dagger$ is the logical conjunction of the interpretations of the rules of P. The logical semantics of a query q is:

$$\mathcal{L}_P(q) = \{c \;|\; (P)^\dagger \models_\mathcal{X} q \rightarrow c\}$$

$\mathcal{L}_P(q)$ is closed by logical implication in this translation: for any set S of logical facts, we denote $\downarrow S = \{c' \mid \exists c \in S, \models_\mathcal{X} c \rightarrow c'\}$, then $\mathcal{L}_P(q) = \downarrow \mathcal{L}_P(q)$.

Let V be the free variables of the initial query (T_0, c_0). Configurations are interpreted as $(T, c)^\dagger = \langle \exists \boldsymbol{x}((T)^\dagger \wedge c) \rangle$ where $\boldsymbol{x} = \mathrm{fv}(T, c) \setminus \mathrm{fv}(T_0, c_0)$: the query variables appear in the observables and are therefore left free in the interpretation. The notation is extended for sets S of configurations: $S^\dagger = \{(T, c)^\dagger \mid (T, c) \in S\}$.

Theorem 1 ([30])**.** *For any left-linear* CHR($\mathcal{X}$) *program P and initial query (T_0, c_0):*

$$\downarrow (\mathcal{O}_P^a(T_0, c_0))^\dagger \subseteq \mathcal{L}_P((T_0, c_0)^\dagger)$$

This formulation gives only a soundness result: $\mathcal{L}_P((T_0, c_0)^\dagger) \subseteq \downarrow (\mathcal{O}_P^a(T_0, c_0))^\dagger$ does not hold in general. For instance, consider the program $\{\mathrm{a} \Leftrightarrow \mathrm{b}.\ \mathrm{a} \Leftrightarrow \mathrm{c}.\}$: $\mathcal{L}_P((\mathrm{a}, \top)^\dagger) = \downarrow\{\mathrm{a} \wedge \mathrm{b} \wedge \mathrm{c}\}$ whereas $\downarrow (\mathcal{O}_P^a(\mathrm{a}, \top))^\dagger = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \top\}$. Weaker soundness and completeness hold: $\mathcal{L}_P((T_0, c_0)^\dagger) = \mathcal{L}_P((\mathcal{O}_P^a(T_0, c_0))^\dagger)$, but completeness is just a consequence of the immediate membership $(T_0, c_0)^\dagger \in (\mathcal{O}_P^a(T_0, c_0))^\dagger$. If all rules of P are simplification rules [3] and if $(\mathcal{O}_P^t(T_0, c_0))^\dagger \neq \varnothing$ (that is to say, P has at least one finite computation), then $\mathcal{L}_P((T_0, c_0)^\dagger) \subseteq \mathcal{L}_P((\mathcal{O}_P^t(T_0, c_0))^\dagger)$: in this case, the property is a direct consequence of $(T_0, c_0)^\dagger \in \mathcal{L}_P((\mathcal{O}_P^t(T_0, c_0))^\dagger)$.

The classical logical semantics is not correct if P is not left-linear. For instance, with the single rule a, a $\Leftrightarrow$ b we have $\mathcal{LL}_P((\mathrm{a},\top)^\dagger) = \downarrow\{\mathrm{a}\wedge\mathrm{b}\}$ while $\downarrow(\mathcal{O}^a_P(\mathrm{a},\top))^\dagger = \downarrow\{\mathrm{a}\} \neq \downarrow\{\mathrm{a}\wedge\mathrm{b}\}$. Furthermore, this logical semantics identifies too many programs (for instance, $\{\mathrm{a} \Leftrightarrow \mathrm{b.a} \Leftrightarrow \mathrm{c.}\}$ and $\{\mathrm{a} \Leftrightarrow \mathrm{b},\ \mathrm{c.}\}$).

To overcome these limitations, a declarative semantics in Girard's linear logic [34] have been developed in [12] for all CHR programs. Built-in constraints over $\mathcal{X}$ are translated using Girard's translation of classical logic in linear logic with the bang operator [34]. A multiset M of constraints is interpreted by a linear tensor of the constraints $M^{\dagger\dagger} = \langle\bigotimes_{c\in M} c\rangle$. For each CHR rule:

$$\big(H\backslash H' \Rightarrow G \;\big|\; B\big)^{\dagger\dagger} = \langle !\forall \boldsymbol{x}(\exists \boldsymbol{y}(G^{\dagger\dagger}) \otimes H^{\dagger\dagger} \otimes H'^{\dagger\dagger} \multimap \exists \boldsymbol{z}(H^{\dagger\dagger} \otimes G^{\dagger\dagger} \otimes B^{\dagger\dagger}_{\mathcal{X}} \otimes B^{\dagger\dagger}_u))\rangle$$

where $\boldsymbol{x} = \mathrm{fv}(H, H')$, $\boldsymbol{y} = \mathrm{fv}(G) \setminus \mathrm{fv}(H, H')$ and $\boldsymbol{z} = \mathrm{fv}(G, B_{\mathcal{X}}, B_u) \setminus \mathrm{fv}(H, H')$.

The linear logical interpretation of a program $(P)^\dagger$ is the linear tensor of the interpretations of the rules of P. The linear logical semantics of a query q is:

$$\mathcal{LL}_P(q) = \{c \;\big|\; (P)^{\dagger\dagger} \models_{LL,\mathcal{X}} q \multimap c \otimes \top\}$$

$\mathcal{LL}_P(q)$ is closed by linear implication: for any set S of logical facts, we denote $\downarrow_\circ S = \{c' \mid \exists c \in S, \models_{LL,\mathcal{X}} c \multimap c' \otimes \top\}$, then $\mathcal{LL}_P(q) = \downarrow_\circ \mathcal{LL}_P(q)$.

Let V be the free variables of the initial query (T_0, c_0). Configurations are interpreted as $(T, c)^{\dagger\dagger} = \langle\exists \boldsymbol{x}(T^{\dagger\dagger} \otimes c)\rangle$ where $\boldsymbol{x} = \mathrm{fv}(T, c) \setminus \mathrm{fv}(T_0, c_0)$: the query variables appear in the observables and are therefore left free in the interpretation. The notation is extended to sets S of configurations: $S^{\dagger\dagger} = \{(T, c)^{\dagger\dagger} \mid (T, c) \in S\}$.

Theorem 2 ([12, 28])**.** *For any* CHR($\mathcal{X}$) *program* P *and initial query* (T_0, c_0)*:*

$$\downarrow_\circ (\mathcal{O}^a_P(T_0, c_0))^{\dagger\dagger} = \mathcal{LL}_P((T_0, c_0)^{\dagger\dagger})$$

Operational Semantics.

A *state* is a tuple denoted $\langle g; b; c\rangle_V$ where g is a goal, b is a built-in constraint, c is a CHR constraint and V is a set of variables. The relation $\equiv_{\mathrm{C}}$ over states is the smallest equivalence relation such that:

- $\langle g; b; c\rangle_V \equiv_{\mathrm{C}} \langle g; b'; c\rangle_V$ for $CT \models b \leftrightarrow b'$;
- $\langle g; b; c\rangle_V \equiv_{\mathrm{C}} \langle g; b; c\rangle_V[y/x]$ for variables $x \notin V$ and $y \notin V \cup \mathrm{fv}(g, b, c)$.

Let $\mathcal{P}$ be the set of pairs of CHR programs and states.

Definition 2 (Naive Operational Semantics [30])**.** A CHR program P is executed along a *transition relation* $\rightarrow_P$ over states:

FIRING RULE

APPLY

$$\frac{\langle H \backslash H' \Leftrightarrow G \mid B\rangle \text{ is a fresh variant of a rule in } P \quad \text{with variables } \boldsymbol{x} \qquad CT \models \forall(b \to \exists \boldsymbol{x}(H \doteqdot h \wedge H' \doteqdot h' \wedge G))}{\langle g; b; h, h', c\rangle_V \to_P \langle B, g; H \doteqdot h \wedge H' \doteqdot h' \wedge G \wedge b; h, c\rangle_V}$$

SOLVING RULES

SOLVE

$$\frac{B \in \mathcal{B} \qquad CT \models B \wedge b \leftrightarrow b'}{\langle B, g; b; c\rangle_V \to_P \langle g; b'; c\rangle_V}$$

INTRODUCE

$$\frac{C \in \mathcal{U}}{\langle C, g; b; c\rangle_V \to_P \langle g; b; C, c\rangle_V}$$

Let q be an initial goal, the *query*. V is defined as $\mathrm{fv}(q)$ and, from the *initial state* $s_0 = \langle q; \top; \varnothing\rangle_V$, a *derivation* is a sequence $s_0 \to_P s_1 \to_P \cdots \to_P s_n$. Such a state s_n is an *accessible state*.

Definition 3 (Linear Logic Semantics [12])**.**
For any built-in constraint $B = \langle B_1 \wedge \cdots \wedge B_n\rangle$,
let $B^\dagger = \langle !B_1 \otimes \cdots \otimes !B_n\rangle$.
For any CHR constraint $C = (C_1, \ldots, C_n)$,
let $C^\dagger = \langle C_1 \otimes \cdots \otimes C_n\rangle$.
For any goal $G = (G_1, \ldots, G_n)$,
let $G^\dagger = \langle G_1^\dagger \otimes \cdots \otimes G_n^\dagger\rangle$.
For any state $S = \langle g; b; c\rangle_V$, let $S^\dagger = \exists \boldsymbol{x}(g^\dagger \otimes b^\dagger \otimes c^\dagger)$,
where $\boldsymbol{x} = \mathrm{fv}(G, B, C) \setminus V$.
The semantics of a rule $r = \langle H \backslash H' \Leftrightarrow G \mid B\rangle$ is
$r^\dagger = \langle !\forall(G^\dagger \otimes H^\dagger \otimes H'^\dagger \multimap \exists \boldsymbol{x}(H^\dagger \otimes B^\dagger))\rangle$
with $\boldsymbol{x} = \mathrm{fv}(B) \setminus \mathrm{fv}(H, H', G)$.
For a program $P = \{r_1, \ldots, r_n\}$, the *linear logic semantics of* P is $P^\dagger = \langle r_1^\dagger \otimes \cdots \otimes r_n^\dagger\rangle$.

Theorem 3 (Soundness & Completeness [12])**.** *Let $CT^\dagger$ be the Girard translation of CT [34], P a* CHR *program and q a query.*

- *(Sound) If s is an accessible state from q in P, then $P^\dagger, CT^\dagger \models \forall(q^\dagger \multimap s^\dagger)$.*
- *(Complete) For every formula c such that $P^\dagger, CT^\dagger \models \forall(q^\dagger \multimap c)$, there is an accessible state s from q in P such that $CT^\dagger \models \forall(s^\dagger \multimap c)$.*

2.1.2 Syntax and Semantics of LLCC

Definition 4 (Linear Constraint System [28])**.** A *linear constraint system* is a pair $(\mathcal{C}, \vdash_{\mathcal{C}})$, where:

- $\mathcal{C}$ is a set of formulas (the *linear constraints*) built from variables $\mathcal{V}$ and the signature Σ, with logical operators: multiplicative conjunction $\otimes$, its neutral 1, existential $\exists$, exponential ! and constant $\top$; $\mathcal{C}$ is assumed to be closed by renaming, multiplicative conjunction and existential quantification;
- $\Vdash_{\mathcal{C}}$ is a binary relation over $\mathcal{C}$, which defines the non-logical axioms.
- $\vdash_{\mathcal{C}}$ is the least subset of $\mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the rules of intuitionistic multiplicative exponential linear logic for 1, $\top$, $\otimes$, ! and $\exists$.

Definition 5 (Syntax with Persistent Asks [41])**.** The syntax for building LLCC *agents* follows the grammar: $A ::= \forall \mathcal{V}^\star(\mathcal{C} \to A) \mid \forall \mathcal{V}^\star(\mathcal{C} \Rightarrow A) \mid \exists \mathcal{V}.A \mid \mathcal{C} \mid A \parallel A$ where $\parallel$ stands for parallel composition, $\exists$ for variable hiding, $\to$ for (transient) ask and $\Rightarrow$ for persistent ask. In the particular case where there are no universally quantified variables in an ask, the notation $(c \to a)$ is preferred to $\forall \varepsilon(c \to a)$.

Agent $\forall \boldsymbol{x}(c \to a)$ suspends until c is entailed then wakes up and does a. Transient asks wake up at most one time. Persistent asks are introduced [41] to replace declarations by agents. The agent $\forall \boldsymbol{x}(c \Rightarrow a)$ can wake up as many times as c is entailed. This behavior makes sense as entailment consumes resources.

Example 2. *Here is the* LLCC *version for dining philosophers [28, 38]. Compared to the* CHR *version, the following code is reentrant: the variable K identifies tokens and let several diners to be run in parallel (a* banquet *[38]) and separation results from the* LLCC *module theory prove that tables cannot steal cutlery from each other [41].*

$$
\begin{array}{l}
\forall N(\mathrm{diner}(N) \Rightarrow \\
\quad \exists K(\forall I(\mathrm{recphilo}(K,I) \Rightarrow \\
\qquad \mathrm{fork}(K,I) \parallel \\
\qquad \exists J.(J \ \mathbf{is}\ (I+1) \bmod N \parallel \\
\qquad\quad (\mathrm{fork}(K,I) \otimes \mathrm{fork}(K,J) \Rightarrow \\
\qquad\quad \ \mathrm{eat}(K,I) \parallel \\
\qquad\quad \ (\mathrm{eat}(K,I) \to \\
\qquad\quad\ \ \mathrm{fork}(K,I) \otimes \mathrm{fork}(K,J)) \parallel \\
\qquad\quad (I < N-1 \to \mathrm{recphilo}(K,J)) \parallel \\
\qquad \mathrm{recphilo(K,0)})))))
\end{array}
$$

This example makes use of non-trivial scopes: variables N, K, I and J are in turn introduced and shared by subsequent asks. The recursive loop $\langle$recphilo$\rangle$ *installs N forks and composes N agents (the philosophers) in parallel. The philosopher between forks I and J is an agent in* LLCC*, whereas she is materialized in 1 by the* CHR *constraint philo (I, J) in order to carry the environment* $\{I, J\}$*.*

A *configuration* is a triple $(X; c; \Gamma)$ where c is a constraint (the *store*), Γ is a multi-set of agents and X is a set of variables (the *hidden variables*). The relation $\equiv_{\mathrm{L}}$ over configurations is the smallest equivalence relation such that:

- $(X; c; a \parallel b, \Gamma) \equiv_{\mathrm{L}} (X; c; a, b, \Gamma)$ for all agents a and b;
- $(X; c; 1, \Gamma) \equiv_{\mathrm{L}} (X; c; \Gamma)$;
- $(X; c; \Gamma) \equiv_{\mathrm{L}} (X; c'; \Gamma)$ for all constraints c, c' such that $c \dashv\vdash_{\mathcal{C}} c'$;
- $(X; c; \Gamma) \equiv_{\mathrm{L}} (X; c; \Gamma)[y/x]$ for all variables $x \in X$ and $y \notin \mathrm{fv}(X, c, \Gamma)$

Let $\mathcal{K}$ be the set of configurations.

Definition 6 (Operational Semantics [28, 41])**.** The *transition relation* $\rightarrow_{\mathrm{L}}$ is the least relation on configurations satisfying the following rules:

FIRING RULES

TRANSIENT ASK

$$\frac{c \vdash_{\mathcal{C}} \exists Y (d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \qquad \forall d' ((c \vdash_{\mathcal{C}} \exists Y (d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash_{\mathcal{C}} d')}{(X; c; \forall \boldsymbol{x} (e \rightarrow a), \Gamma) \rightarrow_{\mathrm{L}} (X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \Gamma)}$$

PERSISTENT ASK

$$\frac{c \vdash_{\mathcal{C}} \exists Y (d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \qquad \forall d' ((c \vdash_{\mathcal{C}} \exists Y (d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash_{\mathcal{C}} d')}{(X; c; \forall \boldsymbol{x} (e \Rightarrow a), \Gamma) \rightarrow_{\mathrm{L}} (X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \forall \boldsymbol{x} (e \Rightarrow a), \Gamma)}$$

SOLVING RULES

HIDING

$$\frac{y \notin X \cup \mathrm{fv}(c, \Gamma)}{(X; c; \exists x.a, \Gamma) \rightarrow_{\mathrm{L}} (X \cup \{y\}; c \otimes d; a[y/x], \Gamma)}$$

TELL

$$\frac{}{(X; c; d, \Gamma) \rightarrow_{\mathrm{L}} (X; c \otimes d; \Gamma)}$$

EQUIVALENCE

$$\frac{\kappa_0 \equiv_{\mathrm{L}} \kappa_0' \rightarrow_{\mathrm{L}} \kappa_1' \equiv_{\mathrm{L}} \kappa_1}{\kappa_0 \rightarrow_{\mathrm{L}} \kappa_1}$$

DECOMPOSITION

$$\frac{}{(X; c; a \parallel b, \Gamma) \rightarrow_{\mathrm{L}} (X; c; a, b, \Gamma)}$$

An agent a is associated with the *initial configuration* $(\varnothing; \top; a)$. *Accessible observables* from a configuration κ are the configurations κ' such that $\kappa \xrightarrow{\star}_{\mathrm{L}} \kappa'$.

Definition 7 (Linear Logic Semantics [28, 41])**.** The translation $(\cdot)^\ddagger$ of LLCC agents into their *linear logic semantics* is defined inductively as follows:

$$(\forall \boldsymbol{x}(c \to a))^\ddagger = \forall \boldsymbol{x}(c \multimap a^\ddagger) \qquad (\forall \boldsymbol{x}(c \Rightarrow a))^\ddagger = !\forall \boldsymbol{x}(c \multimap a^\ddagger) \qquad (\exists x.a)^\ddagger = \exists x(a^\ddagger)$$

$$c^\ddagger = c \qquad\qquad (a \parallel b)^\ddagger = a^\ddagger \otimes b^\ddagger$$

If Γ is a multi-set of agents $(a_1, \ldots, a_n)$, we define $\Gamma^\ddagger = \langle a_1^\ddagger \otimes \cdots \otimes a_n^\ddagger \rangle$. Configurations are translated to $(X; c; \Gamma)^\ddagger = \langle \exists X(c \otimes \Gamma^\ddagger) \rangle$.

The following theorem has first been proved by Fages, Ruet & Soliman [28]. The proof was done for a presentation of LLCC with declarations and without persistent asks. Rémy Haemmerlé showed in its Ph. D thesis that declarations can be internalized in the agents by the introduction of persistent asks: the work was presented in [41] and the full proof is in Rémy's Ph. D, [38].

Theorem 4 (Soundness & Completeness [28, 41, 38])**.** *For all agents a:*

- *(Sound) If κ is an accessible observable from $(\varnothing; \top; a)$, then $a^\ddagger \vdash_{\mathcal{C}} \kappa^\ddagger$.*
- *(Complete) If c is such that $a^\ddagger \vdash_{\mathcal{C}} c$, then there is an accessible observable $(X; d; \Gamma)$ from $(\varnothing; \top; a)$ with $\exists X(d) \vdash_{\mathcal{C}} c$ and agents in Γ are persistent asks.*

2.1.3 Circumscribing non-determinism in CHR and LLCC operational semantics

Whereas non-determinism in firing rules seems to be inherent to the computation model (and is tackled in CHR by the committed-choice strategy and by the refined semantics), the non-determinism in sequencing solving rules can be completely eliminated. This is a classical result for constraint logic programming [45]. We formalize such a result for CHR and LLCC since the precise bisimulation results presented in next sections rely on it.

Let $\to_P^{\mathrm{s}}$ and $\to_P^{\mathrm{f}}$ be the restrictions of $\to_P$ to solving and firing rules respectively. Let $\to_{\mathrm{L}}^{\mathrm{s}}$ and $\to_{\mathrm{L}}^{\mathrm{f}}$ be the similar restrictions for $\to_{\mathrm{L}}$.

We define $\Rightarrow_P^{\mathrm{s}}$ such that $s \Rightarrow_P^{\mathrm{s}} s'$ if and only if $s \xrightarrow{\star}{}_P^{\mathrm{s}} s' \not\to_P^{\mathrm{s}}$. Similarly, $\Rightarrow_{\mathrm{L}}^{\mathrm{s}}$ is such that $\kappa \Rightarrow_{\mathrm{L}}^{\mathrm{s}} \kappa'$ if and only if $\kappa \xrightarrow{\star}{}_{\mathrm{L}}^{\mathrm{s}} \kappa' \not\to_{\mathrm{L}}^{\mathrm{s}}$.

Lemma 1 (Solving rules terminate and are confluent modulo $\equiv$)**.** *For every* CHR *program P, for all state s, there exists s' such that $s \Rightarrow_P^{\mathrm{s}} s'$ and for all s', s'', if $s \Rightarrow_P^{\mathrm{s}} s'$ and $s \Rightarrow_P^{\mathrm{s}} s''$, then $s' \equiv_{\mathrm{C}} s''$.*
For every configuration κ, there exists κ' such that $\kappa \Rightarrow_{\mathrm{L}}^{\mathrm{s}} \kappa'$ and for all κ', κ'', if $\kappa \Rightarrow_{\mathrm{L}}^{\mathrm{s}} \kappa'$ and $\kappa \Rightarrow_{\mathrm{L}}^{\mathrm{s}} \kappa''$ then $\kappa' \equiv_{\mathrm{L}} \kappa''$.

Thus, observed configurations can be restricted to be final for $\rightarrow^{\mathrm{s}}$ (or, equivalently, normalized by $\Rightarrow^{\mathrm{s}}$) without losing derivations. The following lemma is a specialization of the "*Andorra*" principle [81] to the rule selection strategy:

Lemma 2 (*Full solving before firing*)**.** *For every* CHR *program* P*,*

$$\left(\overset{\star}{\rightarrow}_P \cdot \Rightarrow^{\mathrm{s}}_P\right) = \left((\Rightarrow^{\mathrm{s}}_P \cdot \rightarrow^{\mathrm{f}}_P)^\star \cdot \Rightarrow^{\mathrm{s}}_P\right)$$

and, similarly,

$$\left(\overset{\star}{\rightarrow}_{\mathrm{L}} \cdot \Rightarrow^{\mathrm{s}}_{\mathrm{L}}\right) = \left((\Rightarrow^{\mathrm{s}}_{\mathrm{L}} \cdot \rightarrow^{\mathrm{f}}_{\mathrm{L}})^\star \cdot \Rightarrow^{\mathrm{s}}_{\mathrm{L}}\right)$$

The lemma 2 is a corollary of the monotonous selection strategy [38]: intuitively, $\rightarrow^{\mathrm{s}}$ can always be exhausted before applying $\rightarrow^{\mathrm{f}}$.

Lemma 3 (Solving rules preserve declarative semantics)**.** *For every* CHR *program* P*, if* $s \rightarrow^{\mathrm{s}}_P s'$*, then* $s^\dagger \equiv s'^\dagger$*. Similarly, if* $\kappa \rightarrow^{\mathrm{s}}_{\mathrm{L}} \kappa'$*, then* $\kappa^\ddagger \equiv \kappa'^\ddagger$*.*

Therefore, next sections focus on $\Rightarrow$-transitions where $\Rightarrow_P = (\Rightarrow^{\mathrm{s}}_P \cdot \rightarrow^{\mathrm{f}}_P \cdot \Rightarrow^{\mathrm{s}}_P)$, and $\Rightarrow_{\mathrm{L}} = (\Rightarrow^{\mathrm{s}}_{\mathrm{L}} \cdot \rightarrow^{\mathrm{f}}_{\mathrm{L}} \cdot \Rightarrow^{\mathrm{s}}_{\mathrm{L}})$: a $\Rightarrow_P$*-accessible state* from s is a state s' such that $s \overset{\star}{\Rightarrow}_P s'$ and a $\Rightarrow_{\mathrm{L}}$*-accessible observable* from κ is a configuration κ' such that $\kappa \overset{\star}{\Rightarrow}_{\mathrm{L}} \kappa'$. It is worth noticing that a firing occurs at each $\Rightarrow$-transition.

2.2 Translations between sub-languages CSR and flat-LLCC

From now on, we consider the linear constraint system $(\mathcal{C}, \vdash_{\mathcal{C}})$ induced by the constraint theory CT and with atomic CHR constraints as linear tokens. More precisely, $\mathcal{C}$ is the least set of formulas which contains $\top$ and $!B$ for all $B \in \mathcal{B}_0$ and C for all $C \in \mathcal{U}_0$, closed by renaming, multiplicative conjunction and existential quantification. We suppose that $c \Vdash_{\mathcal{C}} d$ if and only if $CT^\dagger \models \forall(c \multimap d)$. The result is a particular form of linear constraint system where non-logical axioms follow from the translation of a classical theory.

Bisimulation is the most popular method for comparing concurrent processes [63], characterizing a notion of strong equivalence between processes. A *transition system* is a tuple $(S, \rightarrow)$ with S a set of states and $\rightarrow$ a binary relation over S. We define the CHR transition system as $(\mathcal{P}, \Rightarrow_{\mathrm{C}})$ where $(P, s) \Rightarrow_{\mathrm{C}} (P', s')$ when $P = P'$ and $s \Rightarrow_P s'$, and the LLCC transition system as $(\mathcal{K}, \Rightarrow_{\mathrm{L}})$.

Definition 8 (Bisimulation)**.** Let $(S_1, \overset{1}{\rightarrow})$ and $(S_2, \overset{2}{\rightarrow})$ be two transition systems. A *bisimulation* is a relation $\sim \subseteq S_1 \times S_2$ such that for all s_1, s_2 such that $s_1 \sim s_2$:

- for all s'_1 such that $s_1 \overset{1}{\rightarrow} s'_1$, there exists s'_2 such that $s_2 \overset{2}{\rightarrow} s'_2$ and $s'_1 \sim s'_2$;
- for all s'_2 such that $s_2 \overset{2}{\rightarrow} s'_2$, there exists s'_1 such that $s_1 \overset{1}{\rightarrow} s'_1$ and $s'_1 \sim s'_2$.

2.2.1 From Constraint Simplification Rules (CSR) to flat-LLCC

Resulting configurations of LLCC Firing Rules enjoy a new store where guards have been consumed. This behavior corresponds to simplification rules in CHR.

Definition 9 (CSR programs[3])**.** A CHR program P is a CSR *program* when all rules of P are simplifications (*i.e.* rules are of the form $\langle H \Leftrightarrow G \mid B. \rangle$).

As far as naive operational semantics and linear-logic semantics are concerned, expressiveness of CHR and CSR is identical. For a rule $r = \langle H \backslash H' \Leftrightarrow G \mid B \rangle$, let $r^{\times} = \langle H, H' \Leftrightarrow G \mid H, B. \rangle$ and for $P = \{r_1, \ldots, r_n\}$, let $P^{\times} = \{r_1^{\times}, \ldots, r_n^{\times}\}$.

Example 3. *Here is leq$^{\times}$ translated from a version of the leq program [70]:*

leq (X, X) ⇔ true .
leq (X, Y) ⇔ number(X), number(Y) | X ⩽ Y.
leq (X, Y), leq (Y, X) ⇔ X = Y.
leq (X, Y), leq (Y, Z) ⇔ leq (X, Y), leq (Y, Z), leq (X, Z).
leq (X, Y), leq (X, Y) ⇔ leq (X, Y).

Proposition 1 (CHR and CSR equivalence)**.** *For every* CHR *program P, we have $\rightarrow_P = \rightarrow_{P^{\times}}$ and $P^{\dagger} \equiv (P^{\times})^{\dagger}$.*

This equivalence only holds for naive CHR semantics. There is probably no natural encoding of the traditional semantics for propagation [26] in LLCC, at least without *ad-hoc* support hard-wired in the constraint system.

Let $r = \langle H' \Leftrightarrow G \mid B. \rangle$ be a simplification rule. $G^{\dagger} \otimes H'^{\dagger}$ and $B^{\dagger}$ are in $\mathcal{C}$, thus the following agent is well-formed: $r^{-\circ} = \langle \forall \boldsymbol{y} (G^{\dagger} \otimes H'^{\dagger} \Rightarrow \exists \boldsymbol{x}. B^{\dagger}) \rangle$, where $\boldsymbol{x} = \mathrm{fv}(B) \backslash \mathrm{fv}(H', G)$ and $\boldsymbol{y} = \mathrm{fv}(H', G)$. For every CSR program $P = \{r_1, \ldots, r_n\}$, the *translation of P in* LLCC is: $P^{-\circ} = \langle r_1^{-\circ} \parallel \ldots \parallel r_n^{-\circ} \rangle$. States $\langle g; b; c \rangle_V$ are translated in $\mathcal{C}$ as well: $\langle g; b; c \rangle_V{}^{-\circ} = g^{\dagger} \otimes b^{\dagger} \otimes c^{\dagger}$.

Example 4. *The leq$^{\times}$ program (Example 3) is translated to the agent* leq$^{-\circ}$*:*

$$
\begin{aligned}
\mathrm{leq}^{-\circ} = & \forall X (\mathrm{leq}(X, X) \Rightarrow 1) \parallel \\
& \forall XY (\mathrm{number}(X) \otimes \mathrm{number}(Y) \otimes \mathrm{leq}(X, Y) \Rightarrow \\
& \qquad X \leqslant Y) \parallel \\
& \forall XY (\mathrm{leq}(X, Y) \otimes \mathrm{leq}(Y, X) \Rightarrow X = Y) \parallel \\
& \forall XYZ (\mathrm{leq}(X, Y) \otimes \mathrm{leq}(Y, Z) \Rightarrow \\
& \qquad \mathrm{leq}(X, Y) \otimes \mathrm{leq}(Y, Z) \otimes \mathrm{leq}(X, Z)) \parallel \\
& \forall XY (\mathrm{leq}(X, Y) \otimes \mathrm{leq}(X, Y) \Rightarrow \mathrm{leq}(X, Y))
\end{aligned}
$$

Since there is no possible confusion between linear tokens and classical constraints, then, by abuse of notations, we omit the ! *operator on $\mathcal{U}_0$ constraints.*

Definition 10 (CSR to LLCC translation)**.** A CSR program P and a query q are translated to the agent $a(P,q) = \langle P^{-\circ} \| q^{\dagger} \rangle$.

Main Result 1 (Bisimilarity)**.** *Let $\sim \subseteq \mathcal{P} \times \mathcal{K}$ be the relation where $(P,s) \sim \kappa$ if and only if $\kappa \equiv_{\mathrm{L}} (X; s^{-\circ}; P^{-\circ})$ with $X = \mathrm{fv}(s) \setminus V$. Then, $\sim$ is a bisimulation.*

Corollary 1 (Semantics preservation)**.** *For* CSR *program P, query q:*

- *if κ is a $\Rightarrow_{\mathrm{L}}$-accessible observable of $a(P,q)$, then $\kappa \equiv (X; c; P^{-\circ})$ and there is a $\Rightarrow_P$-accessible state s from q with $\exists \boldsymbol{x}(s^{-\circ}) \dashv\vdash_{\mathcal{C}} \exists X(c)$, $\boldsymbol{x} = \mathrm{fv}(s) \setminus \mathrm{fv}(q)$;*
- *if s is a $\Rightarrow_P$-accessible state from q, then there is a $\Rightarrow_{\mathrm{L}}$-accessible observable $(X; c; P^{-\circ})$ from $a(P,q)$ such that $\exists \boldsymbol{x}(s^{-\circ}) \dashv\vdash_{\mathcal{C}} \exists X(c)$, where $\boldsymbol{x} = \mathrm{fv}(s) \setminus \mathrm{fv}(q)$.*

2.2.2 From flat-LLCC to CSR

The translation of CSR into LLCC generates agents of the particular form $p \parallel q$, where the sub-agent p is the translation of a CSR program and is therefore a parallel composition of persistent asks without any nested asks, and the sub-agent q is a translation of a query and is therefore reduced to a constraint. Moreover, every ask guard consumes at least a linear token (since CHR heads are non-empty) and asks are closed term (*i.e.* without free variables). Such agents are characterized by the following definition:

Definition 11 (flat-LLCC)**.** *Flat*-LLCC *agents* are restricted to the grammar: $A^{\uparrow} ::= A^{\forall} \parallel \mathcal{C}$ where $A^{\forall} ::= \forall \mathcal{V}^{\star}(\mathcal{C} \Rightarrow \mathcal{C}) \mid A^{\forall} \parallel A^{\forall} \mid 1$ with the following side condition for every ask $\forall \boldsymbol{x}(g \Rightarrow c)$: $g \not\vdash_{\mathcal{C}} g \otimes g$ (consumption) and $\mathrm{fv}(g,c) \subseteq \boldsymbol{x}$.

This subsection is dedicated to establishing the reverse translation, from $A^{\uparrow}$ to CSR. It is worth noticing first that, like a CSR program, an $A^{\uparrow}$-agent essentially transforms constraint stores without introducing new suspensions:

Lemma 4 (Configurations form)**.** *Non-initial $\Rightarrow_{\mathrm{L}}$-accessible configurations from an $A^{\uparrow}$-agent a are $\equiv_{\mathrm{L}}$-equivalent to configurations of the form $(\_; \_; a^{\forall})$.*

The translation from flat-LLCC to CSR should handle the LLCC existential variables which have no counter part in CSR and the splitting between built-in constraints and CHR constraints. Fresh variables should be introduced to translate constraints such as $a(X,Y) \otimes \exists X(b(X,Y))$ into $\langle a(X,Y), b(K,Y) \rangle$ where K is a new local variable. The function $f^{\mathcal{C}}$ translates every constraint in $\mathcal{C}$ to a tuple $(X; B; C)$ where B is a built-in constraint, C a CHR constraint and X a set of

variables local to B and C:

$$\begin{aligned}
f^{\mathcal{C}}(\top) &= (\varnothing; \mathbf{true}; \varnothing) \\
f^{\mathcal{C}}(!B) &= (\varnothing; B; \varnothing) \\
&\quad \text{for all } B \in \mathcal{B}_0 \\
f^{\mathcal{C}}(C) &= (\varnothing; \mathbf{true}; C) \\
&\quad \text{for all } C \in \mathcal{U}_0 \\
f^{\mathcal{C}}(c \otimes d) &= (\sigma_c(X_c) \cup \sigma_d(X_d); \sigma_c(B_c) \wedge \sigma_d(B_d); \\
&\qquad \sigma_c(C_c), \sigma_d(C_d)) \\
&\qquad \text{where } f^{\mathcal{C}}(c) = (X_c; B_c; C_c) \\
&\qquad \text{and } f^{\mathcal{C}}(d) = (X_d; B_d; C_d) \\
&\qquad \text{with } \sigma_c \text{ and } \sigma_d \text{ renaming of } X_c \text{ and} \\
&\qquad X_d \text{ respectively such that} \\
&\qquad \sigma_c(X_c) \cap \mathrm{fv}(\sigma_d(B_d, C_d)) = \varnothing \text{ and} \\
&\qquad \sigma_d(X_d) \cap \mathrm{fv}(\sigma_c(B_c, C_c)) = \varnothing \\
f^{\mathcal{C}}(\exists x(c)) &= (X_c \cup \{x\}; B_c; C_c) \\
&\quad \text{where } f^{\mathcal{C}}(c) = (X_c; B_c; C_c)
\end{aligned}$$

$A^{\forall}$-agents are translated to CSR programs through the function $f^{\forall}$. Translation of asks should take care of clashes with similar renaming as for $\otimes$ in $f^{\mathcal{C}}$:

$$\begin{aligned}
f^{\forall}(\forall \boldsymbol{x}(g \Rightarrow c)) &= \\
&\quad \{\langle \sigma_g(C_g) \Leftrightarrow \sigma_g(B_g) \mid \sigma_c(B_c), \sigma_c(C_c). \rangle\} \\
&\qquad \text{where } f^{\mathcal{C}}(g) = (X_g; B_g; C_g) \\
&\qquad \text{and } f^{\mathcal{C}}(c) = (X_c; B_c; C_c) \\
&\qquad \text{and } \sigma_g \text{ and } \sigma_c \text{ renaming of } X_g \text{ and} \\
&\qquad X_c \text{ respectively such that} \\
&\qquad \sigma_g(X_g) \cap \mathrm{fv}(\sigma_c(B_c, C_c)) = \varnothing \text{ and} \\
&\qquad \sigma_c(X_c) \cap \mathrm{fv}(\sigma_g(B_g, C_g)) = \varnothing \\
f^{\forall}(a \parallel b) &= f^{\forall}(a) \cup f^{\forall}(b) \\
f^{\forall}(1) &= \varnothing
\end{aligned}$$

For every ask $\forall \boldsymbol{x}(g \Rightarrow c)$, $f^{\forall}(\forall \boldsymbol{x}(g \Rightarrow c))$ is a well-formed CHR rule. In particular, the side condition on g ensures that $\sigma_g(C_g) \neq \varnothing$.

$f^{\mathrm{s}}_V : c \mapsto \langle \varnothing; b; c \rangle_V$ maps constraints to states with $(\_; b; c) = f^{\mathcal{C}}(c)$.

Note that all variables in CSR queries are global. The query should hide existentially quantified variables in the top-level constraint c_0 of the agent. We suppose a fresh symbol $\langle \mathrm{start}/n \rangle \in \mathcal{U}_0$ where $n = \#\mathrm{fv}(c_0)$.

Definition 12 (Flat-LLCC to CSR translation)**.** A flat-LLCC agent $\langle a^{\forall} \parallel c_0 \rangle$ is translated to the CHR program $P(a^{\forall} \parallel c_0) = f^{\forall}(a^{\forall}) \cup \{\mathrm{start}(\boldsymbol{v}) \Leftrightarrow B_0, C_0.\}$ and the query $q(a) = (\mathrm{start}(\boldsymbol{v}))$ where $(\_; B_0; C_0) = f^{\mathcal{C}}(c_0)$ and $\boldsymbol{v} = \mathrm{fv}(c_0)$.

Main Result 2 (Bisimilarity). *Let $\sim \subseteq \mathcal{K} \times \mathcal{P}$ be the relation where $\kappa \sim (P, s)$ if and only if there exists a flat-*LLCC *agent $\langle a^\forall \parallel c_0 \rangle$ where $\kappa \equiv_{\mathrm{L}} (X; c; a^\forall)$ and $P = P(a)$ and $s \equiv_{\mathrm{C}} f^{\mathrm{s}}_V(c)$, with $V = \mathrm{fv}(c_0)$. Then, $\sim$ is a bisimulation.*

Corollary 2 (Semantics preservation). *For every flat-*LLCC *agent $a = \langle a^\forall \parallel c_0 \rangle$, let $s_0 = \langle q(a); \top; \varnothing \rangle_V$, $V = \mathrm{fv}(c_0)$, then:*

- *for all $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c; a^\forall)$ from a, there exists $a \Rightarrow_{P(a)}$-accessible state s from s_0 such that $\exists \boldsymbol{x}(s^{-\circ}) \dashv\vdash_{\mathcal{C}} \exists X(c)$;*
- *for all $\Rightarrow_{P(a)}$-accessible state s from s_0, if $s \neq s0$, there exists $a \Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c; a^\forall)$ from a, such that $\exists \boldsymbol{x}(s^{-\circ}) \dashv\vdash_{\mathcal{C}} \exists X(c)$;*

where, in both cases, $\boldsymbol{x} = \mathrm{fv}(s) \setminus V$.

2.2.3 CHR Linear-Logic and Phase Semantics Revisited

Lemma 5 (Identical Semantics). *For every* CSR *program P and query q, $(P^{-\circ})^\ddagger \equiv P^\dagger$ and we have $P^\dagger, CT^\dagger \models \forall(q^\dagger \multimap c)$ if and only if $(P^{-\circ})^\ddagger, q^\dagger \vdash_{\mathcal{C}} c$.*

Relating 3 and 4 supposes to prove that accessible constraints are included in provable constraints (correctness), and conversely (completeness). Thus, the correctness and completeness result amounts to equality between sets, which we make explicit here to prove both ways at the same time.

$$\mathcal{O}_{\mathrm{C}}(P,q) = \left\{(P,s) \in \mathcal{P} \;\middle|\; (q;\top;\varnothing) \xrightarrow{\star}_P s\right\} \qquad \mathcal{O}_{\mathrm{L}}(a) = \left\{\kappa \in \mathcal{K} \;\middle|\; (\varnothing;\top;a) \xrightarrow{\star}_{\mathrm{L}} \kappa\right\}$$
$$\mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P,q) = \left\{(P,s) \in \mathcal{P} \;\middle|\; (q;\top;\varnothing) \overset{\star}{\Rightarrow}_P s\right\} \qquad \mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a) = \left\{\kappa \in \mathcal{K} \;\middle|\; (\varnothing;\top;a) \overset{\star}{\Rightarrow}_{\mathrm{L}} \kappa\right\}$$
$$\mathcal{LL}_{\mathrm{C}}(P,q) = \left\{c \in \mathcal{C} \;\middle|\; P^\dagger, CT^\dagger \models \forall(q^\dagger \multimap c)\right\} \qquad \Downarrow^{\mathrm{s}}_{\mathrm{C}} S = \left\{s \in \mathcal{P} \;\middle|\; \exists s' \in S \Rightarrow^{\mathrm{s}}_{\mathrm{C}} s\right\}$$
$$\mathcal{LL}_{\mathrm{L}}(a) = \left\{c \in \mathcal{C} \;\middle|\; a^\ddagger \vdash_{\mathcal{C}} c\right\} \qquad \Downarrow^{\mathrm{s}}_{\mathrm{L}} S = \left\{\kappa \in \mathcal{K} \;\middle|\; \exists \kappa' \in S \Rightarrow^{\mathrm{s}}_{\mathrm{L}} \kappa\right\}$$
$$\downarrow\!\!\circ\, S = \left\{c \in \mathcal{C} \;\middle|\; \exists c' \in S, c' \vdash_{\mathcal{C}} c\right\}$$

Some results mentioned up to now are summarized in the following table:

For every CSR program P, query q, and flat-LLCC agent a,
- 3: $\downarrow\!\!\circ(\mathcal{O}_{\mathrm{C}}(P,q))^\dagger = \mathcal{LL}_{\mathrm{C}}(P,q)$;
- 4: $\downarrow\!\!\circ(\mathcal{O}_{\mathrm{L}}(a))^\ddagger = \mathcal{LL}_{\mathrm{L}}(a)$;
- 2: $\Downarrow^{\mathrm{s}}_{\mathrm{C}}\mathcal{O}_{\mathrm{C}}(P,q) = \mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P,q)$ and $\Downarrow^{\mathrm{s}}_{\mathrm{L}}\mathcal{O}_{\mathrm{L}}(a) = \mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a)$;
- 3: $(\Downarrow^{\mathrm{s}}_{\mathrm{C}}\mathcal{O}_{\mathrm{C}}(P,q))^\dagger = (\mathcal{O}_{\mathrm{C}}(P,q))^\dagger$ and $(\Downarrow^{\mathrm{s}}_{\mathrm{L}}\mathcal{O}_{\mathrm{L}}(a))^\ddagger = (\mathcal{O}_{\mathrm{L}}(a))^\ddagger$;
- 1: $\mathcal{O}_{\mathrm{C}}(P,q) = \mathcal{O}_{\mathrm{C}}(P^\times,q)$ and $\mathcal{LL}_{\mathrm{C}}(P,q) = \mathcal{LL}_{\mathrm{C}}(P^\times,q)$;
- 1: $(\mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P^\times,q))^\dagger = (\mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a(P^\times,q)))^\ddagger$;
- 5: $\mathcal{LL}_{\mathrm{C}}(P^\times,q) = \mathcal{LL}_{\mathrm{L}}(a(P^\times,q))$

We are now ready to prove 3 again from the other results.

Proof of 3. For every CHR program P and query q:

$$\begin{array}{llll}
\downarrow\!\!\circ(\mathcal{O}_{\mathrm{C}}(P,q))^{\dagger} & \overset{1}{=} \downarrow\!\!\circ(\mathcal{O}_{\mathrm{C}}(P^{\times},q))^{\dagger} & \overset{3}{=} & \downarrow\!\!\circ(\Downarrow^{\mathrm{s}}_{\mathrm{C}}\mathcal{O}_{\mathrm{C}}(P^{\times},q))^{\dagger} \\
& \overset{2}{=} \downarrow\!\!\circ(\mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P^{\times},q))^{\dagger} & \overset{1}{=} & \downarrow\!\!\circ(\mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger} \\
& \overset{2}{=} \downarrow\!\!\circ(\Downarrow^{\mathrm{s}}_{\mathrm{L}}\mathcal{O}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger} & \overset{3}{=} & \downarrow\!\!\circ(\mathcal{O}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger} \\
& \overset{4}{=} \mathcal{LL}_{\mathrm{L}}(a(P^{\times},q)) & \overset{5}{=} & \mathcal{LL}_{\mathrm{C}}(P^{\times},q) \\
& \overset{1}{=} \mathcal{LL}_{\mathrm{C}}(P,q) & & \square
\end{array}$$

The following proposition describes a method to prove unreachability property in CHR using phase semantics, adapted from similar result in LLCC [28].

Proposition 2 (Safety through Phase Semantics [39])**.** *To prove a safety property of the kind $s \not\to_P s'$ for a given* CHR *program P, it is enough to prove that for a well-chosen phase space $\mathbf{P}$ and valuation η compatible with CT and P, there exists an element $a \in \eta(s^{\dagger})$ such that $a \notin \eta(t^{\dagger})$.*

Such a valuation η is compatible with $\vdash_{\mathcal{C}}$ and $(P^{\times})^{\multimap}$ (note that it is immediate to see $(P^{\times})^{\multimap}$ as declarations in the sense of the original presentation of LLCC [28]). Let κ and κ' be the respective images of s and s' by the transformation. Then the element a is such that $a \in \eta(\kappa^{\ddagger})$ and $a \notin \eta(\kappa'^{\ddagger})$. Thus $\kappa \not\to_{\mathrm{L}} \kappa'$ comes from the phase semantics of LLCC. Therefore the property $s \not\to_{P^{\times}} s'$ follows from 1, and is generalizable to P with 1. That proves 2. $\square$

2.3 Ask-lifting: encoding LLCC into CSR

The main result of this section is a translation from LLCC to flat-LLCC which preserves the semantics. Consequently, thanks to 2, we can deduce a semantics-preserving translation from LLCC to CSR. This section begins with a preliminary step introducing an intermediary language LLCC$^{\ell}$ where asks are labeled with linear tokens: these tokens do not change the operational semantics and there is a trivial labeling to transform LLCC programs to LLCC$^{\ell}$ programs. These linear tokens are introduced in order to follow asks through the transitions of the operational semantics, which is used to prove the semantics preservation.

2.3.1 Preliminary step: labeling LLCC-agents

Labeled LLCC agents A^{ℓ} differ from agents A by labels inserted on each ask. In the following definition, labels are arbitrary linear tokens.

Definition 13 (LLCC$^\ell$ agents)**.** The syntax of LLCC$^\ell$ *agents* is given by the following grammar:

$$\begin{aligned} A^\ell ::= & \forall \mathcal{V}^\star(\mathcal{C} \xrightarrow{\mathcal{U}_0} A^\ell) \mid \forall \mathcal{V}^\star(\mathcal{C} \overset{\mathcal{U}_0}{\Rightarrow} A^\ell) \mid \\ & \exists \mathcal{V}.A^\ell \mid \mathcal{C} \mid A^\ell \parallel A^\ell \end{aligned}$$

The transition relation $\to_{\mathrm{L}}$ is lifted to the transition $\to_{\mathrm{LLCC}^\ell}$ for LLCC$^\ell$.

Transient Ask (with labeling)

$$\frac{\begin{array}{c} c \vdash_{\mathcal{C}} \exists Y(d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \\ \forall d'((c \vdash_{\mathcal{C}} \exists Y(d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash d') \end{array}}{\Big(X; c; \forall \boldsymbol{x}(e \xrightarrow{l} a), \Gamma\Big) \to_{\mathrm{LLCC}^\ell} (X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \Gamma)}$$

Persistent Ask (with labeling)

$$\frac{\begin{array}{c} c \vdash_{\mathcal{C}} \exists Y(d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \\ \forall d'((c \vdash_{\mathcal{C}} \exists Y(d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash d') \end{array}}{\Big(X; c; \forall \boldsymbol{x}(e \overset{l}{\Rightarrow} a), \Gamma\Big) \to_{\mathrm{LLCC}^\ell} \Big(X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \forall \boldsymbol{x}(e \overset{l}{\Rightarrow} a), \Gamma\Big)}$$

Agents A are translated to a particular family of labeled agents denoted A^{ℓ_0} with the labeling transformation, which ensures the following conditions: each label carries a distinct symbol taken from a set $\mathcal{P}$ of fresh predicate symbols and the arguments enumerate exactly the free variables of the ask. Such a labeling is simple to obtain as soon as $\mathcal{P}$ is large enough to label each ask of a.

Example 5. *The dining philosophers (2) can be labeled as follows:*

$$\begin{array}{l} \forall N(\mathrm{diner}(N) \overset{p_1}{\Rightarrow} \\ \quad \exists K(\forall I(\mathrm{recphilo}(K, I) \xrightarrow{p_2(K,N)} \\ \qquad \mathrm{fork}(K, I) \parallel \\ \qquad \exists J.(J \textbf{ is } (I+1) \bmod N \parallel \\ \qquad\quad (\mathrm{fork}(K, I) \otimes \mathrm{fork}(K, J) \xrightarrow{p_3(I,J,K)} \\ \qquad\quad \mathrm{eat}(K, I) \parallel \\ \qquad\quad (\mathrm{eat}(K, I) \xrightarrow{p_4(I,J,K)} \\ \qquad\quad \mathrm{fork}(K, I) \otimes \mathrm{fork}(K, J)) \parallel \\ \qquad\quad (I < N - 1 \xrightarrow{p_5(I,J,K,N)} \\ \qquad\quad \mathrm{recphilo}(K, J)) \parallel \\ \qquad \mathrm{recphilo}(\mathrm{K}, 0)))))) \end{array}$$

2.3.2 The ask-lifting transformation

The ask-lifting transformation is defined with two helper functions. $\langle a \rangle^{\mathcal{C}}$ transforms the agent a to constraints where asks become linear tokens. $\langle a \rangle^{\forall}$ puts in parallel every ask occurring in a and the representing token is added to the guard. A persistent ask restores the token, a transient ask consumes it.

The function $\langle \cdot \rangle^{\mathcal{C}} : A^{\ell} \to \mathcal{C}$ is defined inductively as follows:

$$\left\langle \forall \boldsymbol{x}(c \xrightarrow{f(\boldsymbol{t})} a) \right\rangle^{\mathcal{C}} = f(\boldsymbol{t}) \qquad \left\langle \forall \boldsymbol{x}(c \xRightarrow{f(\boldsymbol{t})} a) \right\rangle^{\mathcal{C}} = f(\boldsymbol{t}) \qquad \left\langle \exists x.a \right\rangle^{\mathcal{C}} = \exists x. \left\langle a \right\rangle^{\mathcal{C}}$$

$$\left\langle a \parallel b \right\rangle^{\mathcal{C}} = \left\langle a \right\rangle^{\mathcal{C}} \otimes \left\langle b \right\rangle^{\mathcal{C}} \qquad \left\langle c \right\rangle^{\mathcal{C}} = c$$

The function $\langle \cdot \rangle^{\forall} : A^{\ell_0} \to A^{\forall}$ is defined inductively as follows:

$$\left\langle \forall \boldsymbol{x}(c \xrightarrow{f(\boldsymbol{v})} a) \right\rangle^{\forall} = \forall \boldsymbol{v}\boldsymbol{x}(f(\boldsymbol{v}) \otimes c \xRightarrow{f(\boldsymbol{v})} \left\langle a \right\rangle^{\mathcal{C}}) \parallel \left\langle a \right\rangle^{\forall}$$

$$\left\langle \forall \boldsymbol{x}(c \xRightarrow{f(\boldsymbol{v})} a) \right\rangle^{\forall} = \forall \boldsymbol{v}\boldsymbol{x}(f(\boldsymbol{v}) \otimes c \xRightarrow{f(\boldsymbol{v})} f(\boldsymbol{v}) \otimes \left\langle a \right\rangle^{\mathcal{C}}) \parallel \left\langle a \right\rangle^{\forall}$$

$$\left\langle \exists x.a \right\rangle^{\forall} = \left\langle a \right\rangle^{\forall} \qquad \left\langle a \parallel b \right\rangle^{\forall} = \left\langle a \right\rangle^{\forall} \parallel \left\langle b \right\rangle^{\forall} \qquad \left\langle c \right\rangle^{\forall} = 1$$

The function $\langle \cdot \rangle^{\forall}$ is well-defined: every ask satisfies the side-condition for $A^{\forall}$.

Definition 14 (Ask-lifting)**.** The *agent ask-lifting function* $[\![\cdot]\!]^{\uparrow} : A \to A^{\uparrow}$ transforms the agent a to the agent $[\![a]\!]^{\uparrow} = \langle a^{\ell} \rangle^{\forall} \parallel \langle a^{\ell} \rangle^{\mathcal{C}}$ where a is translated to a^{ℓ} by the labeling defined in 2.3.1 with symbol predicates from a subset $\mathcal{P}$ of $\mathcal{P}_c$ whose predicates do not appear in a. $[\![\cdot]\!]^{\uparrow}$ is well-defined as soon as the set $\mathcal{P}$ is large enough to label agent a.

Main Result 3 (Bisimilarity)**.** *Let a be a labeled* LLCC *agent. Let $\sim \subseteq \mathcal{K} \times \mathcal{K}$ be the relation such that $\kappa \sim \kappa'$ if and only if $\kappa \equiv_{\mathrm{L}} (X; c; \Gamma)$ is $\Rightarrow$-accessible from a and $\kappa' \equiv_{\mathrm{L}} (X; c \otimes \langle \Gamma \rangle^{\mathcal{C}}; \langle a \rangle^{\forall})$. Then, $\sim$ is a bisimilarity.*

Corollary 3 (Semantics preservation)**.** *For every* LLCC *agent a:*

- *for all $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c; \Gamma)$ from a, there is a $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c'; \langle a \rangle^{\forall})$ from $[\![a]\!]^{\uparrow}$ such that $\exists X(c \otimes \langle \Gamma \rangle^{\mathcal{C}}) \dashv\vdash_{\mathcal{C}} \exists X'(c')$;*
- *for all $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c'; \langle a \rangle^{\forall})$ from $[\![a]\!]^{\uparrow}$, there exists a $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c; \Gamma)$ from a and $\exists X(c \otimes \langle \Gamma \rangle^{\mathcal{C}}) \dashv\vdash_{\mathcal{C}} \exists X'(c')$.*

Example 6. *The labeled diner (5) can be lifted as follows:*

$$
\begin{array}{ll}
\forall N(& p_1 \otimes \mathrm{diner}(N) \Rightarrow \\
 & p_1 \otimes p_2(K,N) \otimes \mathrm{recphilo}(K,0)) \parallel \\
\forall IKN(& p_2(K,N) \otimes \mathrm{recphilo}(K,I) \Rightarrow \\
 & p_2(K,N) \otimes \mathrm{fork}(K,I) \otimes \\
 & \exists J(J \textbf{ is } (I+1) \bmod N \otimes p_3(I,J,K) \otimes \\
 & \quad p_5(I,J,K,N))) \parallel \\
\forall IJK(& p_3(I,J,K) \otimes \mathrm{fork}(K,I) \otimes \mathrm{fork}(K,J) \Rightarrow \\
 & p_3(I,J,K) \otimes \mathrm{eat}(K,I) \otimes p_4(I,J,K)) \parallel \\
\forall IJK(& p_4(I,J,K) \otimes \mathrm{eat}(K,I) \Rightarrow \\
 & \mathrm{fork}(K,I) \otimes \mathrm{fork}(K,J)) \parallel \\
\forall IJKN(& p_5(I,J,K,N) \otimes I < N-1 \Rightarrow \mathrm{recphilo(K, J)}) \parallel \\
 & p_1
\end{array}
$$

2.4 Encoding the call-by-value λ-calculus in CHR

The following transformation from pure λ-terms to LLCC is proved correct and complete with respect to the call-by-value semantics of the λ-calculus [41]. Every function, *aka* λ-value, is represented by a variable K. The constraint $\mathrm{apply}(K,X,V)$ represents that V should code the result of the application of the function (coded by) K to the λ-term (coded by) X. Therefore, the transformation of a λ-abstraction $\lambda x.e$ coded by K should be a persistent ask which transforms, for all X and V, the constraint $\mathrm{apply}(K,X,V)$ to the equality constraint between V and the evaluation of $e[t/x]$, where t is the λ-term coded by X. The equality constraint is put at the level of λ-variables. The constraint $\mathrm{value}(K)$ indicates that the λ-term K has been reduced to a value so as to encode the particular call-by-value strategy [61].

Definition 15 (Call-by-value λ-calculus in LLCC [41])**.** For every λ-term e, $[\![e]\!]$ is a function from variables to LLCC agents. $[\![e]\!]$ is described inductively on the structure of e:

- $[\![X]\!](K) = \langle X = K \otimes \mathrm{value}(K) \rangle$
- $[\![\lambda X.e]\!](K) =$
  $\forall XV(\mathrm{apply}(K,X,V) \otimes \mathrm{value}(X) \Rightarrow$
  $[\![e]\!](V) \parallel \mathrm{value}(X))$
- $[\![f\ e]\!](K) = \exists XY(\mathrm{apply}(X,Y,K) \parallel [\![f]\!](X) \parallel [\![e]\!](Y))$

Each ask introduced by this transformation corresponds to a λ-abstraction and this property is preserved by ask-lifting. Therefore, the CSR program obtained by translation has one rule for each λ-abstraction.

We explicit below the direct transformation from λ-terms to CSR. We suppose that the labeling has been prepared directly in λ-terms: λ-abstractions are of the form $\lambda_i X.e$ where i is a unique index.

Definition 16 (Call-by-value λ-calculus in CHR)**.** For every λ-term e, $[e]$ is a function from variables to pairs CHR programs and queries, each component being denoted $[e]^p$ and $[e]^g$. $[e]$ is described inductively on the structure of e as follows.

- $[X](K) = (\varnothing; (X = K, value(K)))$
- $[\lambda X_i.e](K) = ([e]^p(V) \cup \{r\}; p_i(K, \boldsymbol{v}))$
  where $\boldsymbol{v} = \mathrm{fv}(\lambda X.e)$ and
  X and V are fresh variables and
  $r = \langle p_i(K, \boldsymbol{v}), \mathrm{value}(X), \mathrm{apply}(K, X, V) \Leftrightarrow$
  $\quad p_i(K, \boldsymbol{v}), \mathrm{value}(X), [e]^g(V).\rangle$
- $[f\ e](K) = ([f]^p(X) \cup [f]^p(Y);$
  $\quad ([f]^g(X), [e]^g(Y), \mathrm{apply}(X, Y, K)))$
  where X and Y fresh variables

The $p_i(\boldsymbol{v})$ CHR constraints are supposed to be fresh. Then, the CSR program associated to e is $P[e] = [e]^p(R) \cup \{\langle start(R, \boldsymbol{v}) \Leftrightarrow [e]^g(R).\rangle\}$ and the query is $q[e] = \mathrm{start}(R, \boldsymbol{v})$ with $\boldsymbol{v} = \mathrm{fv}(e)$.

It is immediate that the program and the goal produced by the transformation above correspond syntactically to the composition of the three transformations: λ-terms to LLCC (15) to flat-LLCC (14) to CSR (12). Therefore, the transformation preserves the semantics as composition of semantics preserving transformations.

In the case of a CHR encoding, the rule associated to each λ-abstraction can be denoted as a simpagation: $\langle p_i(K, \boldsymbol{v}), \mathrm{value}(X) \backslash \mathrm{apply}(K, X, V) \Leftrightarrow [e]^g(V).\rangle$.

Example 7. *The λ-term $(\lambda_1 X.\lambda_2 Y.X)$ A B is transformed to the rules:*

```
start(R,A,B) ⇔
      p1(F1), apply(F1,A0,F2), apply(F2,B0,R),
      A=A0, value(A0), B=B0, value(B0).
p1(F1), value(X) \ apply(F1,X,F2) ⇔ p2(F2,X).
p2(F2,X), value(Y) \ apply(F2,Y,R) ⇔ X = R, value(R).
```

and the following goal, where the variable R codes the result:

```
| ?- start(R, A, B).
p1(_) value(_X) value(_) value(_X) p2(_, _X)
R = A
```

Chapter 3

Reified Search: search strategies as constraint solving

In Section 3.1, we define constraint satisfaction problems as relational languages, that leads to very general definitions for constraints and propagators: in particular, "global constraints", that is to say constraints that act on a unbounded number of variables, can here be presented as a single constraint, *i.e.* relational language, whereas usual presentations introduce them as families of constraints. Domain representations are introduced in the form of abstractions over the point-wise domain. In Section 3.3, we introduce the ClpZinc language and describe the transformation of search procedures from CLP($\mathcal{X} + \mathcal{H}$) to and/or trees over $\mathcal{X}$ with some tree traversal. In Section 3.3.2, we describe the transformation from those and/or trees to their internalization in CSP($\mathcal{X}$), *i.e.*, into Zinc models with a back-end solver for $\mathcal{X}$. In the subsequent sections, we evaluate this approach on benchmarks of models with specific search strategies, namely: Korf's Square Packing problem in Section 3.4, limited discrepancy search in Section 3.5.1 and symmetry breaking during search in Section 3.5.2. In Section 3.6, we show how it is possible to go beyond tree search procedures by using a simple mechanism of annotations for global store, and specify optimization procedures such as Branch-and-Bound.

3.1 Constraint Satisfaction

3.1.1 Constraints as Relational Languages

Let $\mathcal{U}$ be a (possibly infinite) set, the *universe of values*. The set of finite sequences of values is denoted $\mathcal{U}^* = \bigcup_{n \in \mathbf{N}} \mathcal{U}^n$.

Definition 17. A *valuation* is an element of $\mathcal{U}^*$.

Definition 18. A *constraint* is a subset of $\mathcal{U}^*$.

A constraint is a set of admissible valuations. The set of constraints is therefore $\mathfrak{P}(\mathcal{U}^*)$. Given a constraint $C \subseteq \mathcal{U}^*$, the valuations $\nu = (\nu_1, \ldots, \nu_n)$ that are in C are called the solutions of C. The proposition $(\nu_1, \ldots, \nu_n) \in C$ is usually written $C(\nu_1, \ldots, \nu_n)$. Constraints generalize n-ary relations over $\mathcal{U}$.

Definition 19. A constraint C is *n-ary* if for all $\nu \in C$, we have $\#\nu = n$ (*i.e.*, $C \subseteq U^n$).

Example 8. *The following constraints are present in most constraint solvers.*

- *Equality* $=$ *and difference* $\neq$ *are binary constraints defined regardless of the universe* $\mathcal{U}$*. These constraints are usually written* $\nu_1 = \nu_2$ *and* $\nu_1 \neq \nu_2$ *for* $=(\nu_1, \nu_2)$ *and* $\neq(\nu_1, \nu_2)$ *respectively.*

- *Every* n*-ary function* $f : \mathcal{U}^n \to \mathcal{U}$ *is an* $(n+1)$*-ary functional relation and therefore induces an* $(n+1)$*-ary constraint:* $f = \{(\nu_1, \ldots, \nu_n, \nu_{n+1}) \in \mathcal{U}^{n+1} \mid f(\nu_1, \ldots, \nu_n) = \nu_{n+1}\}$*. The constraint is usually written* $f(\nu_1, \ldots, \nu_n) = \nu_{n+1}$ *instead of* $f(\nu_1, \ldots, \nu_n, \nu_{n+1})$*.*

- *Suppose that a subset* $A \subseteq \mathcal{U}$ *is ordered (typically,* A *is a set of numbers). The usual comparison operators* $<, >, \leqslant, \geqslant$ *over* A *are binary constraints.*

- *If* C *only contains a finite number* m *of solutions, these solutions can be explicitly given in a matrix* $M \in \mathcal{U}^{m \times n}$*. In this case,* C *is usually called the* $\textsc{Table}_M$ *constraint or the* $\textsc{Relation}_M$ *constraint:* $\textsc{Table}_M = \{(\nu_1, \ldots, \nu_n) \in \mathcal{U}^n \mid \exists i, \forall j, M_{i,j} = \nu_j\}$*.*

- *Subsets of* U *are* 1*-ary constraints, usually called* domain *constraints. The notation* $\nu_1 \in C$ *is often preferred over* $C(\nu_1)$ *in this case.*

Suppose $\mathbf{Z} \subseteq \mathcal{U}$*.*

- *The usual arithmetic operators* $+, -, \times, \div$*, as binary functions, are ternary constraints.*

- *The usual congruence is a ternary constraint:* $\textsc{Congruent} = \{(a, b, c) \in U^3 \mid a \equiv b \ (\textbf{\textit{mod}}\ c)\}$*.*

A constraint satisfaction problem describe, for a set $\mathcal{X}$ of variables, the constraints that the instantiations of these variables should satisfy to be solutions to the problem.

Definition 20. A *constraint satisfaction problem* is given by

- a finite set $\mathcal{X}$ of variables;

- an initial family of *domains* $D : \mathcal{X} \to \mathfrak{P}(\mathcal{U})$ for all variables;
- a finite set $\mathcal{C} \subseteq \mathfrak{P}(\mathcal{U}^*) \times \mathcal{X}^*$ of (parameterized) constraints.

The elements $(C, \boldsymbol{x})$ of $\mathcal{C}$ are written $C(\boldsymbol{x})$: $C(x_1, \ldots, x_n)$ denotes the formal application of the constraint C to the variables $(x_1, \ldots, x_n)$.

Definition 21. An *instantiation* is an element of $\mathcal{U}^{\mathcal{X}}$.

Let ν be an instantiation. ν is extended to formal applications: $\nu(C(x_1, \ldots, x_n))$ denotes the proposition $C(\nu(x_1), \ldots, \nu(x_n))$, *i.e.* $(\nu(x_1), \ldots, \nu(x_n)) \in C$.

Definition 22. An instantiation ν is a *solution* when, for all $C(\boldsymbol{x}) \in \mathcal{C}$, we have $\nu(C(\boldsymbol{x}))$.

A constraint satisfaction problem realizes the conjunction of the constraints $\mathcal{C}$ over $\mathcal{X}$ and is usually denoted $C_1(\boldsymbol{x}_1) \wedge \cdots \wedge C(\boldsymbol{x}_n)$, where $\mathcal{C} = \{\boldsymbol{x}_1, \ldots \boldsymbol{x}_n\}$. $\mathcal{X}$ and D are often left implicit: $\mathcal{X}$ is the set of variables that occur in $\mathcal{C}$ and D can usually be inferred from $\mathcal{C}$ through a simple propagation of domain constraints.

Example 9. *Let $\mathcal{G} = (V, E)$ be a graph and $\mathcal{U}$ a set of colors. The following constraint satisfaction problem describes the graph coloring of $\mathcal{G}$ with the colors in $\mathcal{U}$.*

$$\bigwedge_{(u,v)\in E} x_u \neq x_v$$

Example 10. *Given two numbers $a, b \in \mathbf{Z}$, the following constraint satisfaction problem over $\mathcal{U} = \mathbf{Z}$ describes the numbers D that are common divisors of a and b. Supplementary variables A, B, U and V are introduced for the computation.*

$$A \in \{a\} \wedge B \in \{b\} \wedge D \times U = A \wedge D \times V = B$$

The domain constraints $A \in \{a\}$ and $B \in \{b\}$ serve here to pass constants to the last parameters of the $\times$ constraints.

Example 11. *The classical n-queens problem can be described by the following constraint satisfaction problem over $\mathcal{U} = \{1, \ldots, n\}$, where each variable x_i represents the column number of the queen at the ith row (or, dually, the row number of the queen at the ith column).*

$$\begin{array}{ll} & \bigwedge_{1\leqslant i<n} \bigwedge_{i<j\leqslant n} x_i \neq x_j \\ \wedge & \bigwedge_{1\leqslant i<n} \bigwedge_{i<j\leqslant n} x_i + i \neq x_j + j \\ \wedge & \bigwedge_{1\leqslant i<n} \bigwedge_{i<j\leqslant n} x_i - i \neq x_j - j \end{array}$$

where the constraints $x_i + i \neq x_j + j$ and $x_i - i \neq x_j - j$ are defined as excepted.

Definition 23. A constraint C is *global* if for all $n \in \mathbf{N}$, there exists $\nu \in C$ such that $\#\nu \geqslant n$.

Example 12. *The following global constraints are present in most constraint solvers.*

- *The* AllDifferent *constraint [52] states that values are pair-wise distinct.*

$$\textsc{AllDifferent} = \{(u_1, \ldots, u_n) \in U^* \mid n \geqslant 0 \text{ and } u_i \neq u_j \text{ for all } i \neq j\}$$

Suppose $\mathbf{N} \subseteq \mathcal{U}$.

- *The* Element *constraint [43] states that a value v is equal to the kth value of a sequence $u_1, \ldots, u_n$.*

$$\textsc{Element} = \left\{(v, k, u_1, \ldots, u_n) \in \mathcal{U}^{n+2} \mid k \in \{1, \ldots, n\} \text{ and } v = u_k\right\}$$

Example 13. *The rules of the Sudoku game on a grid of size $n^2 \times n^2$ (usually, $n = 3$) can be written as a constraint satisfaction problem.*

$$\begin{array}{ll} & \bigwedge_{0 \leqslant i,j < n^2} x_{i,j} \in \{1, \ldots, n^2\} \\ \wedge & \bigwedge_{0 \leqslant i < n^2} \textsc{AllDifferent}(x_{i,0}, \ldots, x_{i,n^2-1}) \\ \wedge & \bigwedge_{0 \leqslant j < n^2} \textsc{AllDifferent}(x_{0,j}, \ldots, x_{n^2-1,j}) \\ \wedge & \bigwedge_{0 \leqslant i,j < n} \textsc{AllDifferent}(x_{n \cdot i, n \cdot j}, \quad \ldots, \quad x_{n \cdot i, n \cdot j + n - 1}, \\ & \qquad\qquad\qquad\qquad\quad \vdots \qquad\qquad\quad \vdots \\ & \qquad\qquad\qquad\qquad x_{n \cdot i + n - 1, j}, \quad \ldots, \quad x_{n \cdot i + n - 1, n \cdot j + n - 1}) \end{array}$$

The constraint Regular [60] is often presented as a global constraint. In the present settings, it is the family of constraints that are regular languages over a certain alphabet.

Definition 24. A constraint C is *regular* if $C \subseteq A^*$ is a regular language, where $A \subseteq \mathcal{U}$ is an alphabet. C is usually denoted $\textsc{Regular}_{\mathcal{A}}$, where $\mathcal{A}$ is an automaton that recognizes C.

Suppose that $\{\mathbf{0}, \mathbf{1}\} \subseteq \mathcal{U}$.

Definition 25. Let C be a constraint. The *reified constraint* associated to C is

$$C' = \left\{(\nu_0, \nu_1, \ldots, \nu_n) \in \mathcal{U}^{n+1} \mid \nu_0 = \mathbf{1} \text{ if } C(\nu_1, \ldots, \nu_n), \text{ otherwise } \nu_0 = \mathbf{0}\right\}$$

$C'(\nu_0, \nu_1, \ldots, \nu_n)$ is usually denoted $\nu_0 = \mathbf{1} \Leftrightarrow C(\nu_1, \ldots, \nu_n)$.

Example 14. *Providing a reflexive view on the truth value of a constraint, reified constraints are very expressive.*

- *The* negation *of every constraint C can be expressed as a reified constraint.*

$$y \in \{\mathbf{0}\} \wedge (y = \mathbf{1} \Leftrightarrow C(x_1, \ldots, x_n))$$

- *The* disjunction *between every couple of constraints (C, D) can be expressed as a reified constraint.*

$$y \in \{\mathbf{0}, \mathbf{1}\} \wedge z \in \{\mathbf{0}, \mathbf{1}\} \wedge y \neq z \wedge (y = \mathbf{1} \Leftrightarrow C(x_1, \ldots, x_n)) \wedge (z = \mathbf{1} \Leftrightarrow D(x'_1, \ldots, x'_{n'}))$$

  This construction is often referred as constructive disjunction*, by opposition to disjunctions occurring in the search phase.*

- *The* equivalence *between every couple of constraints (C, D) can be expressed as a reified constraint.*

$$y \in \{\mathbf{0}, \mathbf{1}\} \wedge (y = \mathbf{1} \Leftrightarrow C(x_1, \ldots, x_n)) \wedge (y = \mathbf{1} \Leftrightarrow D(x'_1, \ldots, x'_{n'}))$$

  This construction is useful for expressing tunnelling *between several representations of the same problem. For example, given two dual instances of the n-queens problem (example 11) $x_1, \ldots, x_n$ for the rows and $x'_1, \ldots, x'_n$ for the columns respectively, then the tunnelling between the two representations can be expressed as follows.*

$$\bigwedge_{1 \leqslant i,j \leqslant n} (y_{i,j} \in \{\mathbf{0}, \mathbf{1}\} \wedge (y_{i,j} = \mathbf{1} \Leftrightarrow x_i = j) \wedge (y_{i,j} = \mathbf{1} \Leftrightarrow x'_j = i))$$

  It is worth noticing that the variables $y_{i,j}$ induce themselves a third representation: the chess-board matrix where $y_{i,j} = \mathbf{1}$ if there is a queen at the ith row and jth column, otherwise $y_{i,j} = \mathbf{0}$.

3.1.2 Propagators and Consistency

We define is this section propagators as closure operators in the sense of the abstract interpretation theory [**?**]. Such a presentation is not new: it can be found for example in [**?**].

Definition 26. Given a set S, a relation $\sqsubseteq \subseteq S \times S$ is a (partial) *order* when $\sqsubseteq$ is

- *reflexive*: for all $x \in S$, $x \sqsubseteq x$;
- *antisymmetric*: for all $x, y \in S$, if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$;
- *transitive*: for all $x, y, z \in S$, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$.

$(S, \sqsubseteq)$ is called a (partially) ordered set. The *dual* order, $\sqsupseteq$, is the order such that, for all $x, y \in S$, $x \sqsupseteq y$ when $y \sqsubseteq x$.

Example 15. *Let A be a set. The set $\mathfrak{P}(A)$ of all the subsets of A is ordered by inclusion $\subseteq$.*

The following definition relies on the usual vocabulary of the abstract interpretation theory [**?**].

Definition 27. Let $(S, \sqsubseteq)$ be a (partially) ordered set and f an *operator* over S (*i.e.*, $f : S \to S$).

- f is *extensive* (or *continuous*) when for all $x \in S$, $x \sqsubseteq f(x)$;
- f is *monotonic* (or *increasing*) when for all $x, y \in S$, if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$.
- f is *idempotent* when $f \circ f = f$;
- f is a *closure operator* when f is extensive, monotonic and idempotent.

Definition 28. Given a closure operator f, an element $x \in S$ is *closed* by f if x is a fix-point of f (*i.e.*, $f(x) = x$).

Since f is idempotent, x is closed by f if and only if $x \in \Im(f)$.

Definition 29. Let $(S, \sqsubseteq)$ be a (partially) ordered set and A be another set. The set of all maps $f : A \to S$ form a partially ordered set $(S^A, \sqsubseteq^*)$, the *point-wise order*, such that, for all $f, g : A \to S$, $f \sqsubseteq^* g$ when $f(a) \sqsubseteq g(a)$ for all $a \in A$.

Example 16. *For all $n \in \mathbf{N}$, since S^n is isomorphic to S^A with $A = \{1, \ldots, n\}$, the point-wise order induces the partially ordered set $(S^n, \sqsubseteq^*)$. Considering the disjoint union $S^* = \bigcup_{n \in \mathbf{N}} S^n$, the point-wise order also induces the partially ordered set $(S^*, \sqsubseteq^*)$ such that $(\mu_1, \ldots, \mu_m) \sqsubseteq^* (\nu_1, \ldots, \nu_n)$ when $m = n$ and $\mu_i \sqsubseteq \nu_i$ for all $1 \leqslant i \leqslant n$.*

Definition 30. A *sequence of domains* is an element of $\mathfrak{P}(\mathcal{U})^*$.

Definition 31. A *propagator* is an extensive operator over sequences of domains ordered by $\supseteq^*$.

It is worth noticing that the extensivity ensures that for all $D \in \mathfrak{P}(\mathcal{U})^*$, $\#\pi(D) = \#D$.

Definition 32. Given a sequence of domains $D = (D_1, \ldots, D_n)$, the *flattening* of D is the set of valuations $\downarrow D = \{(\nu_1, \ldots, \nu_n) \in \mathcal{U}^* \mid \forall i, \nu_i \in D_i\}$.

Definition 33. Given a constraint C and a sequence of domains D, the *set of solutions of C in D* is the set $\mathrm{sol}_C(D) = C \cap {\downarrow}D$.

sol_C is monotonic: for all sequences of domains D and D', if $D \subseteq^* D'$, then $\mathrm{sol}_C(D) \subseteq \mathrm{sol}_C(D')$.

Definition 34. A propagator π filters a constraint C when it preserves the set of solutions of C (*i.e.*, for all sequence of domains D, $\mathrm{sol}_C(\pi(D)) = \mathrm{sol}_C(D)$).

Definition 35. Given a constraint C, the *canonical propagator for C* is

$$\begin{array}{rccc} \pi_C : & \mathfrak{P}(\mathcal{U})^* & \to & \mathfrak{P}(\mathcal{U})^* \\ & D = (D_1, \ldots, D_n) & \mapsto & D' = (D'_1, \ldots, D'_n) \end{array}$$

where $D'_i = \{\mu_i \in D_i \mid \exists \nu \in \mathrm{sol}_C(D), \mu_i = \nu_i\}$.

π_C is extensive by definition: $D'_i \subseteq D_i$.

3.1.3 Abstractions

Definition 36. An *abstraction* is a closure operator over sequences of domains ordered by $\subseteq^*$.

Definition 37. Given an abstraction α, a sequence of domains $D \in \mathfrak{P}(\mathcal{U})^*$ is *α-consistant* with respect to a constraint C when it is the smallest sequence of domains with respect to $\subseteq^*$ among the sequences of domains D' such that

- D' is closed by α,
- $\mathrm{sol}_C(D) \subseteq \mathrm{sol}_C(D')$.

Property 1. *A sequence of domains $D \in \mathfrak{P}(\mathcal{U})^*$ is α-consistant with respect to a constraint C if and only if for all propagators π that filter C, D is a fix-point for $\alpha \circ \pi$.*

Proof. Suppose that D is α-consistent with respect to C. Then, for all propagators π that filter C, let $D' = \alpha \circ \pi(D)$. We will show that $D = D'$.

- $D \subseteq D'$: We show that D' satisfies the hypotheses of Definition 37 so that the minimality of D concludes. Indeed, D' is closed by α since α is idempotent. Moreover, $\mathrm{sol}_C(D) \subseteq \mathrm{sol}_C(D')$ since π preserves the set of solutions and α is extensive with respect to $\subseteq^*$.
- $D' \subseteq D$: Indeed, $D \supseteq \pi(D)$ since π is extensive, therefore $\alpha \circ \pi(D) \subseteq \alpha(D)$ since α is monotonic, and $\alpha(D) = D$ since D is closed.

Conversely, suppose that, for all propagators π that filter C, $D = \alpha \circ \pi(D)$.

□

3.2 The Language ClpZinc

We propose to use Constraint Logic Programming (CLP) clauses to specify search strategies in Zinc. More precisely, given a constraint system $\mathcal{X}$ (e.g. finite domain constraints) and a CSP model with constraints in $\mathcal{X}$, we consider search procedures that are expressible as the traversal of an and/or tree with constraints over $\mathcal{X}$, *i.e.* an and/or tree where every leaf is either labeled by a constraint in $\mathcal{X}$ or, for dynamic search procedures, labeled by a query to indexicals. In addition, we consider the Prolog primitive constraint system, $\mathcal{H}$, *i.e.* Herbrand terms with unification. The choice of Herbrand terms for representing Zinc data structures makes the language look familiar to Prolog users. Similarly, we fix the CLP strategy as depth-first and left-to-right.

The modeling language Zinc, and its implementation MiniZinc [1], succeeded in becoming a common input format across many solvers in the Constraint Programming community. In Zinc, the search procedure is specified through special annotations that are dedicated to the constraint solver [62] and ignored by the other solvers.

The language ClpZinc is an extension of Zinc where the item solve satisfy ; in models is replaced by a CLP goal of the form ":- *goal*.", and where user-defined predicates are defined by CLP clauses of the form "p($t_1,\ldots,t_n$) :- *goal*.".

Example 17 (Labeling)**.** *The following ClpZinc model implements the search strategy that enumerates all possible values for a given variable in ascending order.*

```
var 0..5: x;
constraint x * x = x + x;

labeling(X, Min, Max) :-
    Min <= Max, (X = Min ; labeling(X, Min + 1, Max)).

:- labeling(x, 0, 5).
output [show(x)];
```

As shown in the following section, this ClpZinc model for the given goal of labeling x between 0 and 5, can be expanded to the following MiniZinc model:

```
var 0..5: x;
constraint x * x = x + x;
var 0..5: X1;
constraint X1 = 0 -> x = 0;
constraint X1 = 1 -> x = 1;
constraint X1 = 2 -> x = 2;
constraint X1 = 3 -> x = 3;
constraint X1 = 4 -> x = 4;
constraint X1 = 5 -> x = 5;
solve :: seq_search([
    int_search([X1], input_order, indomain_min, complete)
  ]) satisfy;
output [show(x)];
```

[1] http://www.minizinc.org/

Definition 38. A *ClpZinc goal* is either

- a constraint,
- a MiniZinc search annotation,
- a call to a user-defined predicate,
- the conjunction (A,B) or the disjunction (A;B) of two goals.

A *ClpZinc clause* is an item of the form $\mathrm{p}(t_1,\ldots,t_n)$:– *goal*. where t_1 and t_n are terms and *goal* is a ClpZinc goal. The goal part can be omitted: "$\mathrm{p}(t_1,\ldots,t_n)$." is a shorthand for "$\mathrm{p}(t_1,\ldots,t_n)$:– true.".

The search annotations of MiniZinc are accessible in goals in order to allow the composition of user-defined strategies with built-in ones. Terms are either logical variables (X, Y, Max, ...), numbers, or compound terms of the form $\mathrm{p}(t_1,\ldots,t_n)$ where t_1, ..., and t_n are terms. Model variables are a special case of compound terms, either atomic (a, b, ...) or array accessors (x[I,J]). Zinc arrays have been unified with Prolog-like lists to ease their enumeration in search strategies.

In CLP($\mathcal{X}+\mathcal{H}$), arithmetic differs from Prolog. Indeed, in accordance with the theory of CLP and unlike most Prolog systems, arithmetic is supposed to be contained in $\mathcal{X}$ and is distinguished from $\mathcal{H}$ terms, *e.g.*, "1 + 1" is undistinguishable from "2" and is not a $\mathcal{H}$ term. In ClpZinc, the different forms of unification, equality, and evaluation predicates that are encountered in Prolog systems (=, #=, **is**, ...) are thus all unified in a unique notion of equality, which is accessible either explicitly with the predicate =, or implicitly when predicate arguments in either $\mathcal{X}$ of $\mathcal{H}$ are unified.

Arithmetic expressions are also extended for accessing the indexicals of the model variables. For instance, the goal M = min(X) assumes that X is a model variable and unifies M with the currently known lower-bound of X. We consider the indexicals min, max, card and dom_nth (for retrieving the nth value in a variable domain). Concretely, an intermediary variable is introduced to receive the value of the indexical and search annotations are emitted for getting them with:

```
annotation indexical_min(var int: target, var int: x);
annotation indexical_max(var int: target, var int: x);
annotation indexical_card(var int: target, var int: x);
annotation indexical_dom_nth(var int: target, var int: x, var int: n);
```

These annotations require to extend the solvers to communicate the indexicals. That is the only change made to the interface of the solvers.

Example 18 (Dichotomic search)**.** *The Zinc indomain_split value selection strategy can be implemented in ClpZinc using indexicals. The predicate dichotomy/3 below expresses the bisection of a variable X that has the initial domain Min . . . Max. The bisection defined in the auxiliary predicate dichotomy/2 is iterated Depth* $= \lceil \log_2 \#X \rceil$ *times to ensure that the domain is reduced to a value on every leaf.*

```
dichotomy(X, Min, Max) :-
    dichotomy(X, ceil(log(2, Max - Min + 1))).

dichotomy(X, Depth) :-
    Depth > 0,
    Middle = (min(X) + max(X)) div 2,
    (X <= Middle ; X > Middle),
    dichotomy(X, Depth - 1).
dichotomy(X, 0).

var 0..5: x;
:- dichotomy(x, 0, 5).
```

The MiniZinc model generated for the given goal is

```
var 0..5: x;
var 0..5: X3; var 0..5: X5; var 0..1: X7;
var 0..5: X4; var 0..5: X6; var 0..5: X2;
var 0..1: X8; var 0..5: X1; var 0..1: X9;
constraint X7 = 0 <-> x <= (X1 + X2) div 2;
constraint X8 = 0 <-> x <= (X3 + X4) div 2;
constraint X9 = 0 <-> x <= (X5 + X6) div 2;
solve :: seq_search([
    indexical_min(X1, x),
    indexical_max(X2, x),
    int_search([X7], input_order, indomain_min, complete),
    indexical_min(X3, x),
    indexical_max(X4, x),
    int_search([X8], input_order, indomain_min, complete),
    indexical_min(X5, x),
    indexical_max(X6, x),
    int_search([X9], input_order, indomain_min, complete)
 ]) satisfy;
```

The next example shows a partial search strategy that is not available using the usual MiniZinc search annotations. This is the interval slitting strategy introduced in [71] for solving Korf's packing problem [49] (explained in Section 3.4) by making a preliminary coarse-grained filtering of the variable domains.

Example 19 (Interval splitting)**.** *The predicate interval_splitting /4 defined below expresses the splitting of the domain of X into intervals of width Step. X is supposed to have the initial domain Min . . . Max.*

```
interval_splitting (X, Step, Min, Max) :-
    Min + Step <= Max, NextX = min(X) + Step,
    (
        X < NextX
```

```
    ;
        X >= NextX,
         interval_splitting (X, Step, Min + Step, Max)
    ).
 interval_splitting (X, Step, Min, Max) :-
    Min + Step > Max.
var 0..5: x;
:- interval_splitting (x, 2, 0, 5).
```

The corresponding MiniZinc model for the given goal is

```
var 0..5: x;
var 0..1: X3; var 0..1: X4; var 0..5: X2;
var 0..5: X1;
constraint X3 = 0 <-> x < X1 + 2;
constraint X3 = 1 -> (X4 = 0 <-> x < X2 + 2);
constraint X3 = 0 -> X4 = 0;
solve :: seq_search([
    indexical_min(X1, x),
    int_search([X3], input_order, indomain_min, complete),
    indexical_min(X2, x),
    int_search([X4], input_order, indomain_min, complete)
 ]) satisfy;
```

3.3 Extending Zinc with CLP Clauses

3.3.1 Partial Evaluation of ClpZinc into And/Or Trees

From now on, let us assume that the initial ClpZinc goals provided in the items ":– *goal*." of the ClpZinc models that we consider, always terminate. That hypothesis should hold even if $\mathcal{X}$ only resolves fully instantiated constraints, as is the case of the static partial evaluator. Verifying termination of logic programs is a classical topic for which many results have been obtained using type systems or abstract interpretation techniques [15]. The description of these techniques is however beyond the scope of this thesis.

Given a constraint system $\mathcal{X}$, the partial evaluation of a CLP($\mathcal{X} + \mathcal{H}$) goal will lead to an and/or tree with constraints over $\mathcal{X}$. The partial evaluator resolves predicate calls, Herbrand constraints and fully instantiated arithmetic constraints, *i.e.*, arithmetic tests. Since, without loss of generality, we settled for a DFS left-to-right CLP evaluation, the and/or trees will be traversed in a similar DFS left-to-right fashion in our examples, but any other traversal order can be treated similarly.

As shown in Figure 3.1 for Example 17, or-nodes are flattened so that nested choices become a single large disjunction. And-nodes are similarly flattened into conjunctions. In the general case, the partial evaluation of the continuation may duplicate constraints with different partial instanciations. For instance, Figure 3.2 shows a simple example of duplication with partial instanciation of the bounding constraint Min <= x, x <= Max.

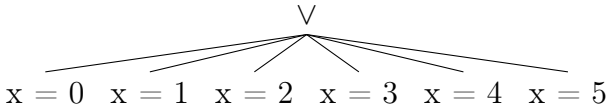

Figure 3.1: And/Or tree derived from Example 17 (Labeling)

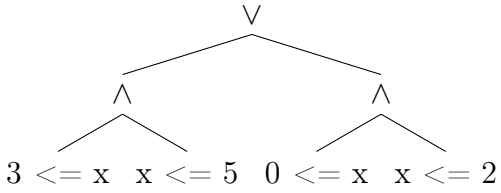

Figure 3.2: And/Or tree for the ClpZinc goal

```
var 0..5: x;
:- (Min=3, Max=5; Min=0, Max=2), Min <= x, x <= Max.
```

However, when the partial evaluation store is left unchanged by a choice (typically, when only constraints in $\mathcal{X}$ are involved), the continuation will remain undeveloped, as shown in Figure 3.3 for Example 18. The and/or tree is in logarithmic size with respect to the size of the domain whereas the fully expanded search tree would be in linear size.

We assume that the partial evaluation terminates and that the resulting and/or tree meets the following conditions:

1. all the variables that appear in constraints are finite-domain variables;
2. all lists are well-formed, in particular the tail of every non-empty list is a list and cannot be a variable since such a variable would be a finite-domain variable according to the previous condition (this ensures that lists can be expanded in Zinc array literals);
3. all annotations except indexicals do not appear below a choice point (*i.e.* their execution is unconditional).

3.3.2 Compiling And/Or Trees into Zinc Reified Constraints

Given a CSP($\mathcal{X}$) model $\mathcal{M}$ and a tree search strategy represented by the traversal of an and/or tree t, the generation of Zinc code proceeds by assigning an additional model variable to every or-node in the tree t, and by emitting search annotations

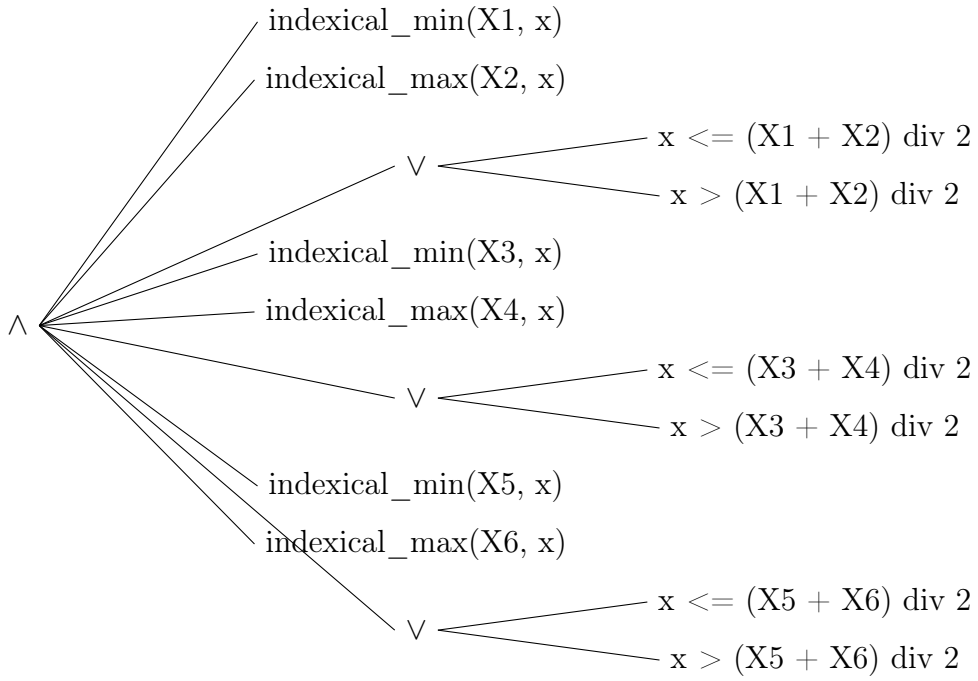

Figure 3.3: And/Or tree for Example 18 (Dichotomic search)

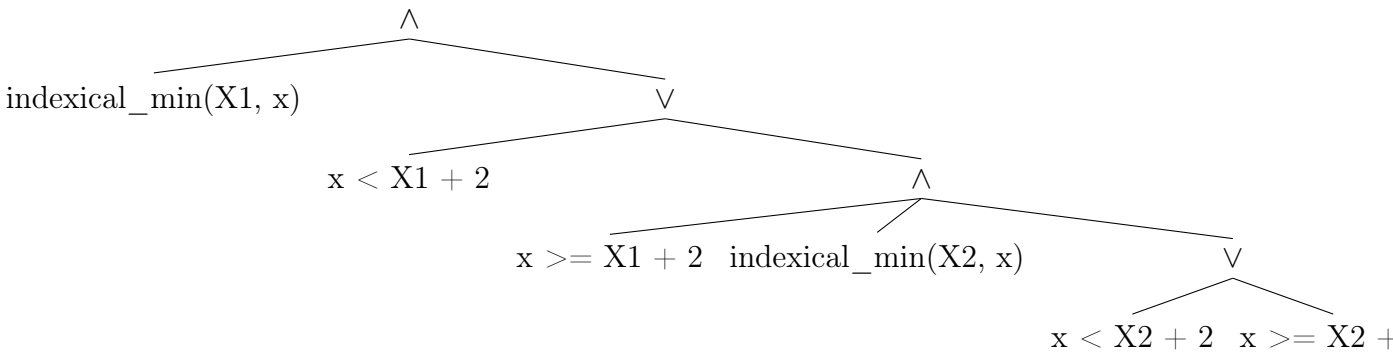

Figure 3.4: And/Or tree for Example 19 (Interval splitting)

that fix the enumeration strategy for these additional variables in a way compatible with the traversal ordering.

In Figure 3.1, the variable X1 is assigned to the root node, with the domain 0..5 corresponding to the arity of the node. As shown in the Zinc model generated for Example 17, each constraint labeling the leaves under this or-node appears in the

model guarded by an implication checking for a particular value of X1. Therefore, when the search annotation int_search enumerates the possible values of X1, these guarded constraints are successively enabled for exploring the different branches of the tree.

More generally, the transformation presented in this chapter can be seen as a constructive proof for the following theorem. We call *fixed enumeration strategies* the search strategies that are reduced to a sequence of variables selected in a fixed order and enumerated with the increasing value selection (indomain_min). For dynamic search strategies, this sequence is possibly interleaved with accesses to indexicals.

Theorem 5. *For every pair $(\mathcal{M}, t)$ where $\mathcal{M}$ is a CSP model and t a tree search strategy described as the traversal of an and/or tree, there exists a model $\mathcal{M}_t$ and a fixed enumeration strategy t' such that the resolution of $(\mathcal{M} + \mathcal{M}_t, t')$ explores the same search tree as $(\mathcal{M}, t)$.*

Proof. First, let us remark that in $\mathcal{M}+\mathcal{M}_t$, the variables and the constraints of $\mathcal{M}$ are left unchanged; only additional model variables accompanied with additional constraints are introduced in $\mathcal{M}_t$.

Let us assume that we have a function ℓ that maps each or-node n of t to a model variable $\ell(n) \in V(\mathcal{M}_t)$, such that for every pair n_1, n_2 of nodes of t, if $\ell(n_1) = \ell(n_2)$, either $n_1 = n_2$ or the lowest common ancestor of n_1 and n_2 is an or-node.

Each constraint c that appears as a leaf of t is translated as a constraint in $\mathcal{M}_t$. Let $n_1, \ldots, n_k$ denote the or-nodes that are traversed by the path π from the root of t to the leaf c and, for every $1 \leqslant i \leqslant k$, let p_i be the rank of the branch taken by π at node n_i. We adopt the convention that branches are numbered from left to right and that the left-most branch has rank 0. Then the following constraint is posted in the MiniZinc model, for translating the leaf c:

$$\text{constraint } \ell(n_1){=}p_1 \ /\backslash \ \ldots \ /\backslash \ \ell(n_k){=}p_k \ {-}{>} \ c;$$

Let $X \in V(\mathcal{M}_t)$ be one of the variables that label or-nodes. The domain of X will be $0..\max\{\mathrm{w}(n) - 1 \mid \ell(n) = X\}$ where $\mathrm{w}(n)$ denotes the width of the or-node n (*i.e.*, the number of branches issued from n). For every or-node n_k such that $\ell(n_k) = X$ that does not reach this maximum, the following additional constraint is posted, where $(n_i, p_i)_i$ denotes the or-path to n_k as above:

$$\text{constraint } \ell(n_1){=}p_1 \ /\backslash \ \ldots \ /\backslash \ \ell(n_{k-1}){=}p_{k-1} \ {-}{>} \ \ell(n_k) < \mathrm{w}(k);$$

We should now establish the connection between the enumeration of the variables that label the or-nodes and the exploration of the and/or tree.

Search annotations have to be emitted to fix the enumeration strategy for the variables $V(\mathcal{M}_t)$. MiniZinc search annotations have a depth-first semantics. To reproduce the semantics of $(\mathcal{M}, t)$, it is thus sufficient in t' to emit the annotations that select the variables in the order where their corresponding nodes are encountered following the traversal of t. The value selection strategy fixes the order in which the sub-branches are explored. For t', it can thus be reduced to the value selection indomain_min that selects the left-most branch, by switching the values if necessary. We thus have that, by construction, $(\mathcal{M} + \mathcal{M}_t, t')$ explores the same search tree as $(\mathcal{M}, t)$. □ □

The two following optimizations are not mandatory but have been measured to give significant performance improvements:

1. to prevent enumerating on X in branches where X does not occur, the following constraint imposes a fixed value to X on these branches.

$$\text{constraint } \Big(\bigwedge_{\substack{(n_i,p_i)_i \\ \ell(n_k)=X}} \ell(n_1)!=p_1 \ \backslash/ \ \ldots \ \backslash/ \ \ell(n_k)!=p_k \Big) \ -> \ X=0;$$

2. in the particular case where a constraint c occurs under an or-node n_k (possibly separated with some and-nodes) and when $\neg c$ occurs in every other branches of n_k, then the following constraint is posted instead (and the constraints corresponding to the leaves $\neg c$ are not posted).

$$\text{constraint } \ell(n_1)=p_1 \ /\backslash \ \ldots \ /\backslash \ \ell(n_{k-1})=p_{k-1} \ -> \ (\ell(n_k)=p_k \ <-> \ c);$$

These simplifications can be seen in the Zinc code generated for Examples 18 and 19.

3.4 Computation Results on Korf's Square Packing Benchmark

In this section, we consider Korf's Optimal Rectangle Square Packing problem [49], *i.e.*, given an integer n $\geqslant 1$, find an enclosing rectangle of smallest area containing n squares from sizes 1×1, 2×2, up to n $\times$ n, without overlap. Helmut Simonis and Barry O'Sullivan proposed a complex dynamic search strategy for that problem in [71], which is interesting to specify and evaluate in ClpZinc. In Fig. 3.5, an optimal solution is drawn for $n = 24$ squares.

First, the model they consider for packing the n consecutive squares in a rectangle of size w $\times$ h can be written in MiniZinc as follows. Since the 1×1 square

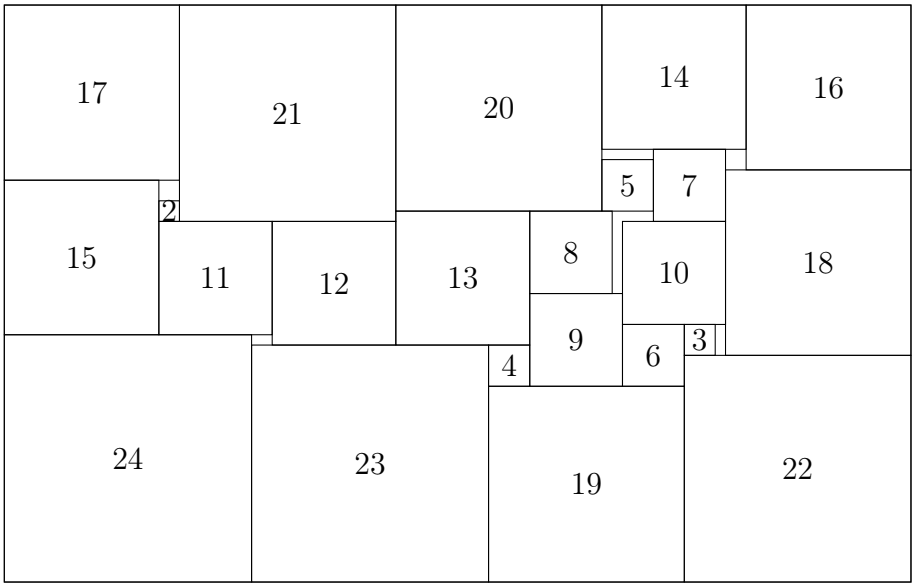

Figure 3.5: Optimal 88×56 solution for Korf's problem with $n = 24$

can always be placed afterward if the area w × h is big enough, the model only considers the remaining n−1 other squares. Two redundant cumulative constraints are introduced. The two last constraints break some symmetries by forcing the largest square to be in the bottom-left quadrant.

```
int: n;

constraint diffn(
   x,y,[i+1|i in 1..n−1],[i+1|i in 1..n−1]
);
constraint cumulative(
   x,[i+1|i in 1..n−1],[i+1|i in 1..n−1],h
);
constraint cumulative(
   y,[i+1|i in 1..n−1],[i+1|i in 1..n−1],w
);

constraint forall(i in 1..n−1)
   (x[i] <= w − i /\ y[i] <= h − i);
constraint x[n−1] <= (w − n + 2) div 2;
constraint y[n−1] <= (h + 1) div 2;
```

Second, the optimization procedure used in [71] enumerates all the possible

sizes w × h for the enclosing rectangle by increasing area. This strategy can be internalized in the model by successively considering all the rectangles up to max_size × max_size, from the minimal area covered by the squares themselves ($\sum_{1 \leqslant i \leqslant n} i^2$) and with bounds on w and h that are described in [71]:

```
int: max_size;
array [1..n−1] of var 1..max_size: x;
array [1..n−1] of var 1..max_size: y;
var 0..max_size: w; var 0..max_size: h;
var 0..max_size * max_size: area;

constraint w * h = area /\ w <= h;
constraint sum([i*i | i in 1..n]) <= area;
constraint w >= 2 * n − 1
  \/ h >= (n * n + n −
        ((w + 1) div 2 − 1) * ((w + 1) div 2 − 1)
    − ((w + 1) div 2 − 1)) div 2;
```

Now, the search strategy of [71] firsts enumerates in x and then in y, considering in each dimension a preliminary interval splitting on the origins of the squares from sizes $n \times n$ to 7×7, and then a dichotomic search on the origins, still by considering the biggest square first. This search strategy is implemented in ClpZinc by enumerating first on area and w to find the rectangle of smallest area first. It is worth noticing that we can combine the user-defined interval splitting strategy defined in Example 19 with the built-in dichotomic search (indomain_split).

```
interval_splitting_list (L, S, Stop) :−
   (S <= Stop ; S > Stop, L = []).
interval_splitting_list ([H | T], S, Stop) :−
   S > Stop,
    interval_splitting (
       H, max(1, (S * 3) div 10) + 1, 0, max_size
    ),
    interval_splitting_list (T, S − 1, Stop).

:− int_search(
       [area, w], input_order, indomain_min, complete
    ),
    reverse(x, RXs), interval_splitting_list (RXs, n, 6),
    int_search(
       RXs, input_order, indomain_split, complete
    ),
    reverse(y, RYs), interval_splitting_list (RYs, n, 0),
```

```
int_search(
    RYs, input_order, indomain_split, complete
).
```

This strategy can be compared to the use of the dichotomic search only, on each dimension, from the biggest to the smallest square, relying on the native indomain_split of MiniZinc. This is indeed a good candidate for the best strategy that can be easily written in MiniZinc without the help of ClpZinc.

```
solve :: seq_search([
    int_search(
        [area, w], input_order, indomain_min, complete
    ),
    int_search(
        [x[n−i] | i in 1..n−1] ++ [y[n−i] | i in 1..n−1],
        input_order, indomain_split, complete
    )
]) satisfy ;
```

To measure the overhead of ClpZinc, we also include the version of dichotomic search relying on the user-defined predicate of Example 18.

```
:− int_search(
    [area, w], input_order, indomain_min, complete
),
reverse(x, Rx), dichotomy_list(Rx, 0, max_size),
reverse(y, Ry), dichotomy_list(Ry, 0, max_size).
```

Table 3.1 shows the results of the native dichotomic search procedure in MiniZinc, of the user-defined dichotomic and interval-splitting search procedure in ClpZinc, solved either Choco or SICStus solvers, and of the original SICStus-Prolog program of [71], all of them running on Intel®Xeon®CPU E5-1620 0 @ 3.60GHz machines. As shown in Table 3.1, the overhead introduced by the reification of the search procedure is quite reasonable, averaging a two-fold slowdown of the program. On the other hand, the reified search enables the encoding of the interval splitting strategy that induces a crucial increase in performance comparable to the results obtained in [71].

That table also shows that the specification in ClpZinc of the dichotomic and interval-splitting search strategy makes it readily available in a variety of solvers for which its implementation was not trivial. The implementation in Choco is the most efficient, followed by SICStus-Prolog, probably due to differences in the implementation of reified constraints.

n	Choco 3			SICStus	
	dichotomic indomain_split	dichotomic ClpZinc	interval split then dichotomic ClpZinc	interval split then dichotomic ClpZinc	interval split then dichotomic Original
16	9232	14402	853	710	340
17	16321	21643	982	450	250
18	422116	570407	7978	9400	4850
19	785080	1051418	6984	11710	4310
20			12572	17330	8970
21			42892	88310	32370
22			208632	303810	153860
23			1340816	2104020	999020
24			2312933	3433410	1481910
25			29201522		10662860
26			142702128		62179600

Table 3.1: Solving times in ms for Korf's problem for strategies implemented in ClpZinc with Choco 3 and SICStus as solvers, compared to the original SICStus program.

3.5 LDS and SBDS as Strategy Transformers in ClpZinc

Since and/or trees are first-class terms in ClpZinc, they can be arguments of ClpZinc predicates to define search strategy transformers. In this section, we illustrate this possibility with the modeling of Limited Discrepancy Search (LDS) [42] and Symmetry Breaking During Search (SBDS) [33] as strategy transformers for labeling or dichotomic search for instance. This technique is closely related to the monadic approach of strategy transformers presented in [69]. The main difference, outside of purely syntactic choices, is that the monadic transformations described in [69] heavily rely on laziness to not expand the trees, whereas in ClpZinc, in order to finally compile towards a CSP, we fully meta-interpret, and therefore expand, the search trees, with some possible benefits thanks to the propagation of search constraints.

3.5.1 Limited Discrepancy Search

LDS can be modeled very simply in ClpZinc using meta-interpretation. Basically the and/or tree is developed but the right turns are counted at the same time,

by increment when going in the right branch of an *or* and by addition of the two branches when going through an *and*:

```
lds(true, L).
lds((A ; B), L) :-
    domain(L0, 0, 1024), domain(D, 0, 1),
    (   D = 0, lds(A, L0)
    ;   D = 1, lds(B, L0)),
    L = D + L0.
lds((A, B), L) :-
    domain(L0, 0, 1024), domain(L1, 0, 1024),
    lds(A, L0), lds(B, L1),
    L = L0 + L1.
lds(B, L) :- builtin(B), B, L = 0.
lds(H, L) :- clause(H, B), lds(B, L).
```

Interestingly, since right turns are counted at the constraint level, the propagation of search constraints may actively reduce the search space, whereas a classical procedural implementation of LDS limits the number of right turns by generate-and-test. The following example demonstrates an *exponential speed-up* thanks to this propagation with respect to a procedural implementation of LDS.

```
var 0..1: x;
var 0..1: y;
array[0..n] of var 0..1: a;

:- int_search(a, input_order, indomain_min, complete),
   lds(((x = 0; x = 1), (y = 0; y = 1)), 0), x != y.
```

Whereas a procedural implementation would explore the 2^n possible assignments for a before detecting that the model is unsatisfiable within the reduced search space, the inconsistency is immediately detected in the MiniZinc model generated by ClpZinc.

```
var 0..1: x;
var 0..1: y;
array[0..n] of var 0..1: a;
constraint x = 0;
constraint y = 0;
constraint x != y;
solve :: seq_search([
    int_search(a, input_order, indomain_min, complete)
  ]) satisfy;
n = 1000;
```

3.5.2 Symmetry Breaking During Search

Symmetry Breaking During Search [8, 33] is a general method that transforms a search tree so as to remove symmetric branches from enumeration. Each time the

search backtracks from enumerating solutions with given a search constraint c, the other search branch considers $\neg c$ and also all the symmetric constraints $\sigma(\neg c)$ for symmetries σ compatible with search constraints already posted. This schema is implemented in the predicate below, supposing a predicate cut_symmetry that adds the symmetric negations for a given constraint.

```
sbds(top, _).
sbds(or(A, B), Path) :-
    (   A = constraint(C, A0),
        (   C, sbds(A, [C | Path])
        ;   cut_symmetry(C, Path), sbds(B, Path))
    ;   A \= constraint(_, _),
        (sbds(A, Path) ; sbds(B, Path))).
sbds(constraint(C, T), Path) :- C, sbds(T, [C | Path]).
:- search_tree(labeling_list(queens, 1, n), T),
   sbds(T, []).
```

The predicate search_tree constructs the search tree associated with the and/or tree of a CLP goal by meta-interpretation (full code in Appendix 3.8).

3.6 Beyond Tree Search Strategies

Some search strategies require to iterate a search tree several times with a memory passed from one branch to another. That is typically the case for optimization methods like branch-and-bound where the best score reached up to now is remembered from one iteration to another of the underlying search strategy, or for shaving, where one step of propagation is performed and undone in order to select the best one. In languages like Prolog, such methods are implemented with the help of a global state, most commonly stored within the fact database (with assert and retract). We propose two additional annotations for search in MiniZinc to handle global state.

```
annotation store(var bool: c, string: id, array[int] of var int: src);
annotation retrieve(string: id, array[int] of var int: target);
```

The semantics of store(cond, id, source) is to remember, if cond is true, the current values of the sequence of variables source into the global state identified as id. The store annotation does nothing if cond is false, such that the assignation to id is skipped outside the computation branch that involves this assignation. The parameter cond does not appear in ClpZinc: it is implicitly fixed to the guard associated to the path leading to the node where the annotation appears in the and/or tree. The semantics of retrieve(id, target) is to assign the values

previously remembered into the global state identified as id into the sequence of variables target.

As shown below, these two simple annotations allow the specification of branch-and-bound optimization in ClpZinc. Once again, for such strategy one might also use the native maximize annotation of MiniZinc, but as far as we know, more complex iterative procedures like shaving or enumerating solutions, using previously found ones in the search (whether to guide it or to limit it), cannot be natively written in MiniZinc.

```
maximize(G, S, Min, Max) :-
    domain(I, Min, Max + 1), domain(Best, Min, Max),
    domain(Fail, 0, 1),
    domain(A, 0, 1), domain(B, 0, 1), domain(C, 0, 1),
    (Fail = 0 -> A != B /\ B != C /\ A != C),
    store("bb_best", [Min, 0]),
    labeling(I, Min, Max + 1),
    retrieve("bb_best", [Best, Fail]),
    ( Fail = 0, store("bb_best", [Best, 1]),
      S > Best, G, store("bb_best", [S, 0]),
      labeling(A, 0, 1), labeling(B, 0, 1)
    ; Fail = 1, I = Max + 1, S = Best, G).

minimize(G, S, Min, Max) :-
    domain(Dual, Min, Max), Dual = Max - S + Min,
    maximize(G, Dual, Min, Max).
```

Note that in order to make this branch-and-bound procedure possible, the gap between failures at the search and at the constraint level has to be bridged. Using the incompleteness of arc-consistency, the reified constraint imposing that A, B and C are all different allows us to fail at will in the success branches (Fail = 0) by labelling A and B. There is also an optimization in the above code where the upper bound on the score is used in the Fail = 1 branch as some kind of *cut*: all attempts after the first failure will be immediately discarded, except the last one where appropriate values for variables will be rebuilt by running the goal G again.

3.7 ClpZinc Models of First-Fail and Middle-out Strategies

The full code for the first-fail variable selection strategy is as follows. Basically, the first_fail predicate selects the variable with smallest domain of the list of variables given as first argument.

```
first_fail ([H | T], X) :-
    (
        check_card_greater_than(T, card(H)),
        X = H
    ;
        first_fail_tail (T, X)
    ).

first_fail_tail (Vars, X) :-
    select (X, Vars, Other_vars),
    CardX = card(X),
    CardX > 1,
    check_card_greater_than(Other_vars, CardX).

check_card_greater_than([], _).

check_card_greater_than([H | T], Min) :-
    CardH = card(H),
    (CardH < Min -> CardH = 1),
    check_card_greater_than(T, Min).
```

The full code for the middle-out value selection strategy is as follows: The middle_out predicate selects the value in the middle of the X variable's domain, removes it from its domain and iterates upper bound times over it.

```
var 0..10 : a;
var 0..10 : b;
var 0..10 : c;

constraint a != b;

middle_out(X, N) :-
    N > 0,
    Middle = dom_nth(X, card(X) div 2),
    (
        X = Middle
    ;
        X != Middle,
        middle_out(X, N - 1)
    ).

middle_out(X, 0).
```

```
:- middle_out(a, 10).

output [ show(a) ];
```

3.8 ClpZinc Model of Symmetry Breaking During Search

The full code for SBDS on the N-queens example, including the transformation of the and/or tree to a search tree during meta-interpretation.

```
int: n;
array[1..n] of var 1..n: queens;

include "globals.mzn";
include "prolog.plz";

constraint all_different(queens);
constraint all_different([queens[i] + i | i in 1..n]);
constraint all_different([queens[i] - i | i in 1..n]);

labeling(X, Min, Max) :-
    Min <= Max,
    (
        X = Min
    ;
        labeling(X, Min + 1, Max)
    ).

labeling_list([], Min, Max).
labeling_list([H | T], Min, Max) :-
    labeling(H, Min, Max),
    labeling_list(T, Min, Max).

builtin(true).
builtin(false).
builtin(_ = _).
builtin(_ <= _).
builtin(_ > _).
```

```
sequence_tree(top, B, T) :-
    search_tree(B, T).

sequence_tree(bot, _T, bot).

sequence_tree(or(A0, B0), T, or(A1, B1)) :-
    sequence_tree(A0, T, A1),
    sequence_tree(B0, T, B1).

sequence_tree(constraint(C, T0), T, constraint(C, T1)) :-
    sequence_tree(T0, T, T1).



search_tree((A, B), T) :-
    search_tree(A, TA),
    sequence_tree(TA, B, T).

search_tree((A ; B), T) :-
    search_tree(A, TA),
    search_tree(B, TB),
    (
        TA == false,
        TB == false,
        T = bot
    ;
        TA == false,
        TB \= false,
        T = TB
    ;
        TA \= false,
        TB == false,
        T = TA
    ;
        TA \= false,
        TB \= false,
        T = or(TA, TB)
    ).

search_tree(B, T) :-
```

```
    builtin(B),
    (
        B == true,
        T = top
    ;
        B == false,
        T = bot
    ;
        B \= true,
        B \= false,
        T = constraint(B, true)
    ).

search_tree(H, T) :-
    findall(B, clause(H, B), L),
    L \= [],
    disjunction(L, D),
    search_tree(D, T).


disjunction([], false).
disjunction([X], X).
disjunction([A, B | T], (A; C)) :-
    disjunction([B | T], C).

sbds(top, _).

sbds(or(A, B), Path) :-
    (
        A = constraint(C, A0),
        (
            C,
            sbds(A, [C | Path])
        ;
            cut_symmetry(C, Path),
            sbds(B, Path)
        )
    ;
        A \= constraint(_, _),
        (
```

```
            sbds(A, Path)
        ;
            sbds(B, Path)
        )
    ).

sbds(constraint(C, T), Path) :-
    C,
    sbds(T, [C | Path]).

:- search_tree(labeling_list(queens, 1, n), T), sbds(T, []).


cut_symmetry(C, Path) :-
    (
        C = (queens[I] = J),
        (
            Path = [],
            queens[I] != n - J + 1
        ;
            Path \= []
        )
    ;
        C \= (queens[I] = J),
        true
    ).

output [
        show(queens) ++ "\n"
    ] ++ [
        if j = 1 then "\n" else "" endif ++
        if fix(queens[i]) = j then
            show_int(2,j)
        else
            "__"
        endif
    | i in 1..n, j in 1..n
    ] ++ ["\n"];

n = 3;
```

Chapter 4

Angelic Programming: from compound guards to atomic kernel for LLCC

4.1 Angelic Semantics and Derivation nets

4.1.1 Process calculi

Definition 39. A *process algebra* $(\mathcal{L}, \otimes, \mathbf{1})$ is a free abelian monoid where

- the set $\mathcal{L}$ is the *language*, its elements a_i are the *agents*,
- the composition law $\otimes$ is the *parallel composition*.

The main point of this definition is introducing the parallel composition as operator for building processes as multisets of *atomic agents* in the style of the *chemical abstract machine* framework CHAM [9].

Definition 40. An *atomic agent* is an agent which is neither $\mathbf{1}$, nor of the form $a_1 \otimes a_2$, with $a_1, a_2 \in \mathcal{L}$.

Notation 1. For any set S, let $\mathcal{M}(S)$ be the set of multisets on S. For every $m \in \mathcal{M}(S)$, the *characteristic application* of m is denoted $\mathbf{1}_m : S \to \mathbf{N}$. The support of a multiset $m \in \mathcal{M}_{\mathrm{f}}(S)$ is $\{s \in S \mid m(s) \geqslant 1\}$. The set of finite multisets on S is denoted $\mathcal{M}_{\mathrm{f}}(S)$, this is the set of multisets m with finite support. For every $s \in S$, $\mathbf{1}_m(s)$ is the *power* of s in m. For any set S' and any application $f : S \to S'$, we note $f(m)$ the image multiset of m by f, with the characteristic application $\mathbf{1}_{f(m)} : s' \in S' \to \sum_{s \in f^{-1}(s')} \mathbf{1}_m(s)$.

Let $\mathcal{L}_0$ be the set of atomic agents. The fact that a process algebra is a free abelian monoid is characterised by the following proposition.

Proposition 3. *The application* $\bigotimes : \begin{array}{ccc} \mathcal{M}_{\mathrm{f}}(\mathcal{L}) & \to & \mathcal{L} \\ \{\!|m_1 \otimes \cdots \otimes m_n|\!\} & \mapsto & m_1 \otimes \cdots \otimes m_n \end{array}$ *is well-defined and its restriction on* $\mathcal{M}_{\mathrm{f}}(\mathcal{L}_0)$ *is one-to-one.*

Proof. Associativity and commutativity justify the definition and $\mathcal{M}_{\mathrm{f}}(\mathcal{L}_0)$ is one-to-one because the process algebra is free. □

Hence, every agent is a "*soup*" of atomic agents.

Definition 41. A *process calculus* $(\mathcal{L}, \otimes, \mathbf{1}, \longrightarrow)$ is given by

- a process algebra $(\mathcal{L}, \otimes, \mathbf{1})$,
- a *reduction relation* $\longrightarrow \subset \mathcal{L} \otimes \mathcal{L}$ which is compatible for parallel composition: for all $a, a', b \in \mathcal{L}$, if $a \longrightarrow a'$ then $a \otimes b \longrightarrow a' \otimes b$.

The compatibility of $\longrightarrow$ with respect to $\otimes$ captures the monotonic property of the calculus with respect to the constraint store in concurrent constraint programming [28] and, more generally, the *chemical law* of the chemical abstract machine framework.

Example 20. *Let* $(C, \Vdash_C)$ *be a linear constraint system. Let* $\mathcal{A}$ *be the set of* LLCC(C) *agents and let* $\equiv$ *be the structural equivalence between agents.* $(\mathcal{A}/\equiv, \| , \mathbf{1}, \longrightarrow)$ *is a process calculus.*

4.1.2 Nets for sets of reductions

Definition 42. An *oriented hypermultigraph* $(V, E, \mathbf{i})$ is given by two disjoint sets V and E, and an application $\mathbf{i} : V \times E \cup E \times V \to \mathbf{N}$ where

- the elements v_i of V are the *vertices*,
- the elements e_j of E are the *edges*,
- the application $\mathbf{i}$ gives the *incidences* between vertices and edges.

Remark 1. Equivalently, an oriented hypermultigraph can be defined as an oriented bipartite graph (V', E') where the edges are weighted by natural numbers $w : E' \to \mathbf{N}$. The correspondence is given by $V' = V \cup E$ and $E' = \{(a, b) \in V \times E \cup E \times V \mid \mathbf{i}(a, b) \geqslant 1\}$, where the weight of each edge is $w = \mathbf{i}|_{E'}$.

In an oriented graph (V', E'), every vertice $v \in V'$ has a set of predecessors ${}^\bullet v = \{v' \mid (v', v) \in E'\}$ and a set of successors $v^\bullet = \{v' \mid (v, v') \in E'\}$. When the graph is weighted by $w : E' \to \mathbf{N}$, predecessors and successors are naturally

considered as multisets, the power of each vertice is given by the weight of the edge linking it. In other words, the characteristic applications of ${}^\bullet v$ and $v^\bullet$ are given by

$$\mathbf{1}_{{}^\bullet v} : v' \in V' \mapsto \begin{cases} w(v', v) & \text{si } (v', v) \in E' \\ 0 & \text{sinon} \end{cases}$$

$$\mathbf{1}_{v^\bullet} : v' \in V' \mapsto \begin{cases} w(v, v') & \text{si } (v, v') \in E' \\ 0 & \text{sinon} \end{cases}$$

It is worth noticing that in the case of bipartite graphs where vertices are partitioned into two classes without edges between vertices from the same class, ${}^\bullet v$ and $v^\bullet$ associates to every vertice a (multi)set of vertices from the other class. We use the same notations for multihypergraphs.

Notation 2. Let $\mathcal{H} = (V, E, \mathbf{i})$ be an oriented hypermultigraph. For every vertice $v \in V$, let ${}^\bullet v$ and $v^\bullet$ be the multiset of edges which are predecessors and successors respectively of v in $\mathcal{H}$. Symmetrically, for every edge $e \in E$, let ${}^\bullet e$ and $e^\bullet$ be the multiset of vertices which are predecessors and successors respectively of e in $\mathcal{H}$.

Let $(\mathcal{L}, \otimes, \mathbf{1}, \longrightarrow)$ be a process calculus.

Definition 43. A *derivation net* $(V, E, \mathbf{i}, \ell_V)$ is given by an oriented hypermultigraph $(V, E, \mathbf{i})$ and an application $\ell : V \to \mathcal{L}$ where

- every vertice $v \in V$ is labelled by an agent $\ell(v) \in \mathcal{L}$
- every edge $e \in E$ is such that $\bigotimes \ell({}^\bullet e) \longrightarrow \bigotimes \ell(e^\bullet)$.

Let $(V, E, \mathbf{i}, \ell_V)$ be a derivation net.

Definition 44. A *marking* m is a multiset of vertices.

Definition 45. The *reduction relation* $\longrightarrow$ is the binary relation $\longrightarrow \subseteq \mathcal{M}_{\mathrm{f}}(V) \times \mathcal{M}_{\mathrm{f}}(V)$ such that $m \longrightarrow m'$ when there exists an edge $e \in E$ satisfying

- ${}^\bullet e \subseteq m$, that is to say for every $v \in V$, $\mathbf{1}_{{}^\bullet e}(v) \leqslant \mathbf{1}_m(v)$,
- $m' = m \ominus {}^\bullet e \oplus e^\bullet$, that is to say for every $v \in V$, $\mathbf{1}_{m'}(v) = \mathbf{1}_m(v) - \mathbf{1}_{{}^\bullet e}(v) + \mathbf{1}_{e^\bullet}(v)$.

Proposition 4. *For all markings m and m', if $m \longrightarrow m'$ then $\bigotimes \ell(m) \longrightarrow \bigotimes \ell(m')$.*

Proof. If $m \longrightarrow m'$, then by definition of the reduction relation, there exists an edge $e \in E$ such that ${}^\bullet e \subseteq m$ and $m' = m \ominus {}^\bullet e \oplus e^\bullet$. By definition of the edges of a derivation net, we have $\bigotimes \ell({}^\bullet e) \longrightarrow \bigotimes \ell(e^\bullet)$. Let $m_0 = m \ominus {}^\bullet e$. By compatibility of $\longrightarrow$ with $\otimes$, we have $\bigotimes \ell({}^\bullet e) \otimes \bigotimes m_0 \longrightarrow \bigotimes \ell(e^\bullet) \otimes \bigotimes m_0$, that is to say $\bigotimes \ell(m) \longrightarrow \bigotimes \ell(m')$. □

In this section, derivation nets are first introduced for Constraint Simplification Rules, the fragment of CHR without propagation. Sharing strategies are introduced to algorithmically build derivation nets that reduce scheduling nondeterminism. Derivation nets are then generalized to the full CHR language through the separation of CHR store between linear and persistent constraints, in the sense of the $\omega_!$ semantics [13].

An (oriented) *multigraph* is a pair $(V; \mathbf{i})$ where V is a set of *vertices* and $\mathbf{i} : V \times V \to \mathbf{N}$ is an *incidence function* giving the weight of the *edge* between each pair of vertices (with the convention that identifies the absence of edge with an edge of weight 0). Equivalenty, $\mathbf{i}$ is a multiset of binary edges in $V \times V$. For each vertex v, the multiset of *prevertices* ${}^\bullet v$, vertices that lead to v, is defined by the characteristic function $u \mapsto \mathbf{i}(u, v)$ and the multiset of *postvertices* $v^\bullet$, vertices that come from v, is defined by the characteristic function $u \mapsto \mathbf{i}(v, u)$. A multigraph is *bipartite* if V is the disjoint union of two sets $V_1 \uplus V_2$ such that $\mathbf{i}(v, v') = 0$ for all $v, v' \in V_i$ for $i = 1$ or 2. An (oriented) *multihypergrap*h is a tuple (V, E, i) such that $(V \uplus E, i)$ is a bipartite multigraph: V is the set of the *vertices of the multihypergraph* and E are the *hyperarcs*. For each hyperarc $e \in E$, ${}^\bullet e$ is the set of *input vertices* of e and $e^\bullet$ is the set of *output vertices* of e. A *labeled multihypergraph* is a tuple (V, E, i, ℓ) such that (V, E, i) is a multihypergraph and $\ell : V \uplus E \to A$ is a mapping from vertices and hyperarcs to an alphabet of *labels* A.

4.1.3 Derivation nets for Constraint Simplification Rules (CSR)

Given a language for built-in constraints $\mathcal{L}_b$ equipped with a constraint theory CT and a language for user-defined constraints $\mathcal{L}_u$, a CSR program is a set of constraint simplification rules.

Definition 46. A *constraint simplification rule* has the form

$$n@H \Leftrightarrow G|B_b, B_u$$

where n is the *name* of the rule, the *head* H is a multi-set of user-defined constraints, the *guard* G is a built-in constraint, and the *body* is a conjunction of a built-in constraint B_b and a multi-set of user-defined constraints B_u.

Consider the following rules describing the calculus of a two-dimensional scalar product with a concurrent product.

Example 21 (Concurrent two-dimensional scalar product)**.**

init @ scalar(X1, Y1, X2, Y2, P) ⇔
product(X1, X2, X), product(Y1, Y2, Y), sum(X, Y, P) .
product @ product(A, B, C) ⇔
V ***is*** *A ∗ B, value(C, V).*
sum @ sum(A, B, C), value(A, VA), value(B, VB) ⇔
V ***is*** *VA + VB, value(C, V) .*

There are essentially two possible derivations in the abstract semantics from a query scalar(X1, Y1, X2, Y2, P), revealing scheduling non-determinism, depending upon which of the two products is evaluated first: product(X1, X2, X) or product(Y1, Y2, Y).

We introduce derivation nets to describe sharing strategies which quotient these scheduling choices.

Definition 47. A derivation net for a CSR program P is a labeled multi-hypergraph $(V, E, \mathbf{i}, \ell)$, where the vertices V are labeled with built-in or user-defined constraints and the hyperarcs E are labeled with rule names ($\ell : V \uplus E \to \mathcal{L}_b \uplus \mathcal{L}_u \uplus \mathcal{N}$), such that for each hyperarc $e \in E$, there exists a rule $\langle n@H \Leftrightarrow G|B_b, B_u\rangle \in P$ and a renaming ρ for fresh variables occurring in the rule with

- $\ell(e) = n$,
- $\ell({}^\bullet e) = H\rho \uplus G'$,
- $\ell(e^\bullet) = B_b\rho \uplus B_u\rho \uplus G'$.

with G' a logical consequence of G under the hypotheses of the theory CT.

The following derivation net shows the quotiented derivation path for the scalar product.

Derivation nets can be equipped with a Petri-net semantics: vertices can be viewed as places marked with tokens that give their number of occurrences in the constraint store. Hyperarcs give the transitions between the places. Compared to the interpretation of CHR programs in Petri-nets [11] that interprets programs themselves as (a colored extension of) Petri-nets independently from the execution, the nets considered here give interpretations for partial executions of programs and grow as long as the execution continues.

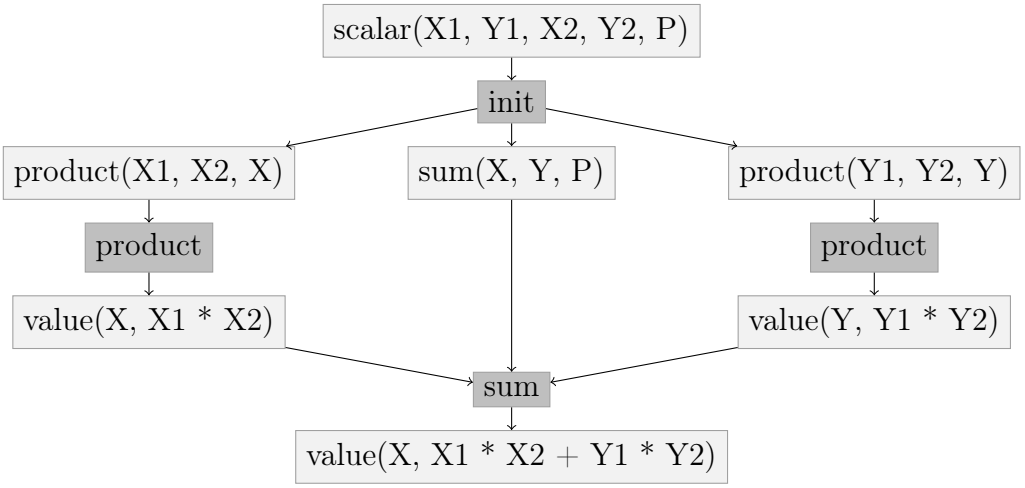

Figure 4.1: Derivation net for the query scalar (X1, Y1, X2, Y2, P) with the scalar product program (Example 21)

Definition 48. A *marking* is a multiset of vertices. *Derivations* are given by a binary relation $\rightarrow_{\mathrm{d}}$ between markings such that $m \rightarrow_{\mathrm{d}} m'$ if there exists a hyperarc e such that ${}^{\bullet}e \subseteq m$ (*i.e.*, for all v, ${}^{\bullet}e(v) \leqslant m(v)$) and for all v, $m'(v) = m(v) - {}^{\bullet}e(v) + e^{\bullet}(v)$.

Markings are multisets of user constraints and built-in constraints: as such, they can be identified to CHR configurations.

Theorem 6 (Correction)**.** *If there exists a derivation $m \rightarrow_{\mathrm{d}} m'$, then there exists a transition in the abstract semantics from the configuration m to the configuration m'.*

Moreover, a derivation net can always be extended with a new hyperarc for any possible transition, leading to new vertices according to the goal of the associated rule. By iterating this construction, it is possible to define a potentially infinite derivation net representing all the possible transitions.

Theorem 7 (Completeness)**.** *For any initial configuration m, there exists a (possibly infinite) derivation net such that if m' is a configuration accessible from m in the abstract semantics, then m' is a marking accessible from m by derivation.*

Angelic execution consists therefore in the iterative construction of such a complete derivation net, keeping only the hyperarcs which are involved in a reachable marking from the initial configuration. Since the exploration is potentially infinite, the exploration should be done in breadth first to give an equal chance of execution to every computation path.

4.1.4 Sharing strategies

Derivation nets do not structurally force any sharing to reduce scheduling non-determinism. This is typically the case for *simpagation* rules: a simpagation rule is of the form $n@H_1 \backslash H_2 \Leftrightarrow B$ where H_1 is a *persistent head*. Such a rule has the same logical interpretation as $n@H_1, H_2 \Leftrightarrow H_1, B$. Two firings of simpagation rules with a common persistent head can lead to two computation paths whether which rule is fired before the other.

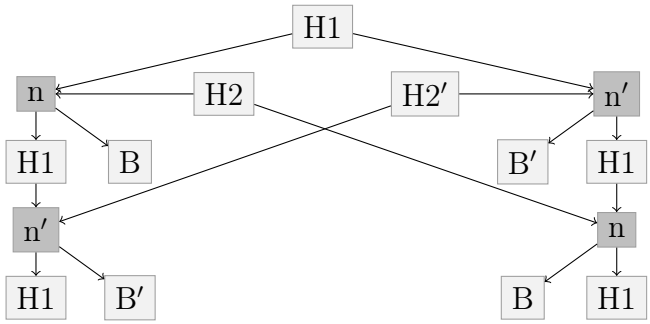

Figure 4.2: Derivation net without sharing for the query H1, H2, H2' with two simpagation rules n @ H_1 H_2 ⇔B and n' @ H_1 H_2' ⇔B'

This non-determinism can be reduced considering the derivation net where each simpagation rule is a hyperarc such that the vertex of the persistent head is the same both for input and output.

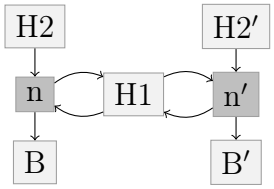

Figure 4.3: Derivation net with sharing for the query H1, H2, H2' with two simpagation rules n @ H_1 H_2 ⇔B and n' @ H_1 H_2' ⇔B'

The iterative construction of such a derivation net can be done by sharing all equal user constraints to the same vertex: interpreting the derivation net as a Petri net, testing the reachability of a hyperarc reduces to testing the reachability in a Petri net, which is decidable [55] but computationaly expensive [53].

Proposition 5. *The complete derivation net where all hyperarcs are reachable and all equal user constraints are shared to the same vertex can be iteratively constructed by solving an EXPSPACE-complete problem for each new hyperarc.*

However, in the context of derivation nets where all cycles are trivial, that is to say that every cycle consist in only one hyperarc having some vertices both as input and output, then there exists a log-linear algorithm to decide the reachability. The case of trivial cycles is particularily important as those cycles appear naturally when considering simpagation rules.

- Preparation: at each new vertex creation v0, computes a table $t(v_0)$ which associates each ancestor vertice v (outside trivial cycles) to its potential immediate successor hyperarc e (there is at most one!): $t(v_0) : v \mapsto e$ With balanced binary trees, logarithmic time cost for each vertex.
- To check if a binary hyperarc e_0 between v_0 and v_1 introduces a conflict:
  1. choose one of the predecessor vertex, say v_0, (preferably the one with least ancestors)
  2. let $t \leftarrow t(v_1) + (v_1 \mapsto e_0)$
  3. begin with $v \leftarrow v_0$,
  4. for each predecessor vertex v' of each predecessor hyperarc e of v,
  5. if $t(v')$ is defined, succeeds if $t(v') = e$ or v' in trivial cycle, else fails,
  6. if not, let $t \leftarrow t + (v' \mapsto e)$ and recursively go to 4 for $v \mapsto v'$.

In worst case, logarithmic cost (table search) for each ancestor.

In practice, either hyperarcs between neighbours ($t(v_0)$ is often defined) or hyperarcs between a vertex and a top-level ask (with few ancestors).

This algorithm gives a polynomial construction for completing the derivation net at each execution step.

4.1.5 Derivation nets in presence of Propagation Rules

As far as the abstract semantics is concerned, propagation rules $n@H \Rightarrow G|B_b, B_u$ are shortcuts for $n@H \Leftrightarrow G|H, B_b, B_u$: the head is restored after firing the rule. This non-consumption leads to trivially infinite computation paths that can be avoided with the $\omega_!$ semantics [13]. In terms of derivation nets, distinguishing the set of vertices between linear and persistent constraints make construction rules of derivation nets being refined to prevent this trivial non-termination. The strict output vertices of a hyperarc are marked as persistent if and only if all the input vertices are persistent or if it is a propagation (that is to say, if all input vertices are in a trivial cycle).

4.2 Angelic Programming

4.2.1 Head Decomposability

In the abstract semantics, a CHR rule can only observe the presence of a user constraint, not the absence: firing is monotonic relatively to the store. As a consequence, if observations are restricted to the side effects of the firing of some rules, silent partial consumption of the head of these rules cannot be observed. Such an observable is motivated by the fact that the body of fired rules is the place where side effects can happen. In particular, multiple headed rules can be rewritten in 2-headed rules by the introduction of fresh intermediary user constraints (carrying the context variables if any).

The rule

$$a,\ b,\ c,\ d \Leftrightarrow \text{print}(\text{"side effect"})$$

and the set of rules

$$a \Leftrightarrow f1$$
$$f1,\ b \Leftrightarrow f2$$
$$f2,\ c \Leftrightarrow f3$$
$$f3,\ d \Leftrightarrow \text{print}\ (\text{"side effect"})$$

are equivalent provided that f1, f2, f3 are fresh user constraints that do not appear elsewhere neither in the program nor in the initial goal.

This equivalence does not hold in general with a committed-choice scheduler since premature consumptions of a and b can prevent other rules to be fired even if c and d never appear. On the contrary, premature consumptions in angelic settings will only lead to blocking computation branches that will not prevent other branches to be explored. This property can benefit to the implementation: only 2-headed rules have to be considered. More precisely, all CHR rules can be translated to rules with two heads where one of them is an intermediary user constraint.

Such a translation makes trivial cycles of simpagations become non-trivial: the algorithm presented above can nevertheless be adapted in this case, since all hyperarcs involved in the cycle only introduce intermediary user constraints. Therefore, a hyperarc cannot be unreachable due to the consumption of such constraints by another rules.

4.2.2 Controlling the Angelism

User constraints can be introduced to explicitely sequence the execution of rules. The following program produces as side-effect a unspecified permutation of a, b, c when launched with the goal start.

```
start ⇔ a, b, c.
a ⇔ print("a").
b ⇔ print("b").
c ⇔ print("c").
```

The order can be fixed by the introduction of fresh intermediary constraints to mark the step of the sequence (carrying the context variables if any).

```
start ⇔ s0, a, b, c.
a, s0 ⇔ s1, print("a").
b, s1 ⇔ s2, print("b").
c, s2 ⇔ print("c").
```

Operationaly, such an explicit sequencing forces the derivation net to have a linear path instead of branching hyperarcs. Formally, the sequence is coded declaratively in the logic instead of being left to the conventional implementation of the comma sequence operator.

4.3 Applications

4.3.1 Angelic CHR∨

Angelic execution can be seen as a search among scheduling. Therefore, CHR∨ rules where there can be multiple bodies, leading to a search for a successful one, can be encoded as multiple rules with the same head.

The rule N @ H ⇔ G | B1 ; ...; Bn. is encoded as the set of n rules N1 @ H ⇔ G | B1., ..., Nn @ H ⇔ G | Bn. Angelic execution ensures that consequences of B1, ..., Bn are explored.

4.3.2 Meta-interpreters

The decomposability of heads allow CHR meta-interpreters to be conveniently written. Suppose that a rule is coded with a user constraint rule(N @ H ⇔ G | B) where H is a list of heads and with a proper encoding for the body B where user constraints are marked with the functor ucstr, then the following rules code a meta-interpreter.

```
first_head @ rule(_N @ H ⇔G, B), ucstr(H0) ⇔
   copy_term((H, G, B), ([H0 | T0], G0, B0)) |
   match(T0, G0, B0).
matching_end @ match([], G, B) ⇔call(G) |
   call(B).
matching_cont @ match([H | T], G, B), ucstr(H) ⇔
```

match(T, G, B).

This meta-interpreter uses the decomposability of rules with match as intermediary user constraint.

Example 22. *The scalar product example (Example 21) is encoded into the following constraints. A derivation net for the meta-interpretation of this program is given in Fig. 4.4.*

```
rule( init  @ [scalar(X1, Y1, X2, Y2, P)] ⇔ true  |
  ucstr(product(X1, X2, X)),
  ucstr(product(Y1, Y2, Y)),
  ucstr(sum(X, Y, P))).
rule(product @ [product(A, B, C)] ⇔ true  |
  V is A * B,
  ucstr(value(C, V))).
rule(sum @ [sum(A, B, C), value(A, VA), value(B, VB)] ⇔true  |
  V is VA + VB,
  ucstr(value(C, V))).
```

4.3.3 User-defined Indexation

Thanks to the dependency book-keeping operated by derivation nets, rules can reformulate user constraints to another form while keeping the dependency relation between the original user constraint and the reformulation. For instance, suppose that the underlying implementation only makes indexing on the principal functor of the user constraint arguments. The following rule decomposes a nested term in a user constraint into another user constraint with this term as root argument.

c(f(X)) ⇔ cf(X) .

Then consuming cf(X) in another rule is equivalent to consuming c(f(X)): therefore, rules having heads matching on c(X) in general and rules having heads matching on cf(X) in particular can coexist and consume observationnaly the same user constraints.

4.3.4 CHR with Ask and Tell

The Constraint Handling Rules language CHR was introduced nearly two decades ago as a declarative language for defining constraint solvers by multiset rewriting rules with guards assuming some built-in constraints [30]. The CHR programming paradigm resolves implementing a constraint system into the declaration of guarded rewriting rules that transform a store of constraints into a solved form

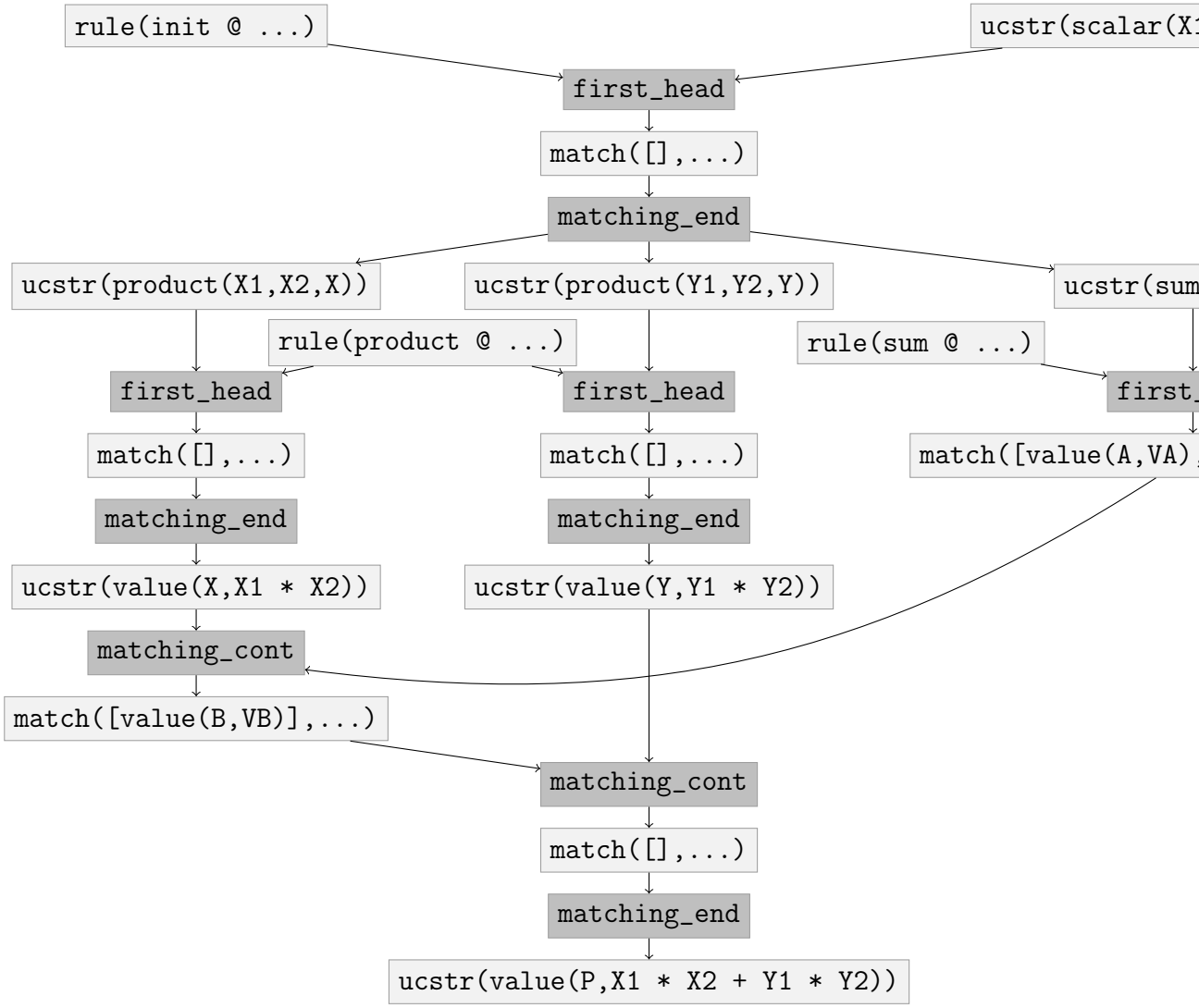

Figure 4.4: Meta-interpretation of the query ucstr(scalar(X1,Y1,X2,Y2,P)) with the meta-interpreted program given in Example 22

allowing to decide satisfiability. Each transformation is supposed to preserve the satisfiability of the system. The solved form, reached when no more transformation can be applied, is unsatisfiable if it contains the constraint "**false**", and is operationally satisfiable otherwise. One important, but not mandatory, property of these transformations is *confluence* which means that the solved form is always independent of the order of application of the rules, and is in fact a *normal form* for the initial constraint store [2].

Since then, CHR has evolved to a general purpose rule-based programming language [30] with some extensions such as for the handling of disjunctions [5] or for introducing types [19]. However, one main drawback ofCHR is the absence of *modularity*. Once a constraint system is defined in CHR with some built-in constraints, this constraint system cannot be reused in another CHR program taking the defined constraints as new built-in constraints. The reason for this difficulty is that a CHR program defines a satisfiability check but not the *constraint entailment* check that is required in guards.

Previous approaches to this problem have studied conditions under which one can derive automatically an entailment check from a satisfiability check. In [67] such conditions are given based on the logical equivalence:

$$\mathcal{X} \models c \rightarrow d \text{ if and only if } \mathcal{X} \models (c \wedge d) \leftrightarrow c$$

In this chapter, we propose a *modular* version of CHR, called CHR with *ask* and *tell*, and denoted CHRat[1]. This paradigm is inspired by the framework of concurrent constraint programming [65, 44]. The programming discipline proposed in CHRat for programming modular constraint solvers is to enforce, for each user-introduced constraint, the definition of simplification and propagation rules to rewrite *ask* control tokens into *entailed* control tokens. Solvers for *asks* and *tells* are required for the implementation of built-in constraint system [27]; the discipline we propose consists in the internalization of this requirement in the CHR solver itself. A guard constraint genericconstraint (argumentvector) in a rule instance R is *operationally entailed* in a constraint store containing the control token ask(K, genericconstraint (argumentvector)) when its solved form contains the token entailed (K, genericconstraint (argumentvector)), where K is a new variable used to associate these control tokens to the rule R. This allows us to program arbitrarily complex entailment checks with rules, instead of event-driven imperative programs [25]. With this programming discipline, CHRat constraints can be reused both in rules and guards in other components to define new solvers.

[1] CHRat and examples of modular definitions of hierarchies of constraint solvers are available at http://contraintes.inria.fr/~tmartine/chrat

CHRat **Components for leq/2 and min/3**

We first consider the classical CHR program defining an ordering constraint.

Example 23. *The satisfiability solver is defined by the following four rules. The first three rules translate the axioms for ordering relations, and the last rule eliminates duplicates in leq constraints.*

$leq(X,X) \Leftrightarrow true.$
$leq(X,Y),\ leq(Y,X) \Leftrightarrow X = Y.$
$leq(X,Y),\ leq(Y,Z) \Rightarrow leq(X,Z).$
$leq(X,Y) \setminus leq(X,Y) \Leftrightarrow true.$

In CHRat, a constraint solver must define rules for checking entailment. The entailment checking rules for leq(X, Y) rewrite the token ask(K, leq(X, Y)) into the token entailed(K, leq(X, Y)). *K* is a variable which associates the control tokens to the rule instances that generated them in order to avoid token capture by other rules with the same guards. In this example, since the store is transitively closed, checking leq(X, Y) is directly observable in the store with a rule if $X \neq Y$. The reflexive case is handled with an additional rule.

leq(X,Y)\ask(K,leq(X,Y)) ⇔ entailed(K,leq(X,Y)).
ask(K,leq(X,X)) ⇔ entailed(K,leq(X,X)).

The satisfiability solver and the entailment solver together define a CHRat component for the constraint leq(X, Y). The modularization of CHRat relies on a simple atom-based component name separation mechanism: exported CHR-constraints are prefixed with the component name and internal CHR-constraints are prefixed so as to avoid collisions [40]. Such a component can be used to define new solvers using the leq(X, Y) constraint both in rules and guards.

Example 24. *For instance, a component for the minimum constraint min(X,Y,Z), stating that Z is the minimum value among X and Y, is definable as follows:*

component min_solver.
import leq/2 from leq_solver.
export min/3.
$min(X,Y,Z) \Leftrightarrow leq(X,Y) \mid Z{=}X.$
$min(X,Y,Z) \Leftrightarrow leq(Y,X) \mid Z{=}Y.$
$min(X,Y,Z) \Rightarrow leq(Z,X),\ leq(Z,Y).$

$min(X,Y,Z) \setminus ask(K,\ min(X,Y,Z)) \Leftrightarrow entailed(K,\ min(X,Y,Z)).$
$ask(K,\ min(X,\ Y,\ X)) \Leftrightarrow leq(X,\ Y) \mid entailed(K,\ min(X,Y,Z)).$
$ask(K,\ min(X,\ Y,\ Y)) \Leftrightarrow leq(Y,\ X) \mid entailed(K,\ min(X,Y,Z)).$

Transformation to a Flat CHR Program

In CHR, the guards are restricted to built-in constraints [30]. CHRat programs are transformed into CHR programs with a program transformation which:

- removes all the user defined constraints from guards;
- renames the predicates ask(c (...)) in ask_c(K,...) and entailed(c (...)) in entailed_c(K,...) with an extra argument for the association variable K;
- introduces for each CHRat rule an id$n$ CHR constraint which links the association variable with the instances of the variables of the head.

For example, the min CHRat solver is transformed to the following CHR program:

```
min(X,Y,Z)\ask_min(K,X,Y,Z)⇒entailed_min(K,X,Y,Z).
min(X,Y,Z)⇒ask_leq(K,X,Y),id1(K,X,Y,Z).
id1(X,Y,Z,K),min(X,Y,Z),entailed_leq(K,X,Y)⇔Z=X.
min(X,Y,Z)⇒ask_leq(K,Y,X),id2(K,X,Y,Z).
id2(X,Y,Z,K),min(X,Y,Z),entailed_leq(K,Y,X)⇔Z=Y.
min(X,Y,Z)⇒leq(Z,X),leq(Z,Y).
ask_min(X,Y,X,K)⇒ask_leq(K0,X,Y),id3(K0,X,Y,K).
id3(K0,X,Y,K),ask_min(K,X,Y,X),entailed_leq(K0,X,Y)⇔
                entailed_min(K,X,Y,X).
ask_min(X,Y,Y,K)⇒ask_leq(K0,Y,X),id4(K0,X,Y,K).
id4(K0,X,Y,K),ask_min(X,Y,Y,K),entailed_leq(K0,Y,X)⇔
                entailed_min(K,X,Y,Y).
```

Existentially Quantified Variables in Guards

In CHRat, as in CHR, variables that appear in a guard without appearing in the head of the rule, need a special treatment. For instance, consider the following CHRat rule for eliminating non minimal elements:

```
number(A),number(B) ⇔min(A, B, C) | number(C).
```

The ask-solver defined above will never detect the entailment of the guard.

In CHRat, the support for existentially quantified variables in guards is performed with control tokens exists(K,V1,...,Vn) where the Vi's are existentially quantified variables in the guard and K is the rule association variable. The rules for ask(min) of example 24 must thus be completed with extra rules for instantiating the existentially quantified variables as follows:

```
ask(K,min(X,Y,Z)),exists(K,Z) ⇔ leq(X,Y) |
        Z=X, entailed(K, min(X,Y,Z)).
```

ask(K,min(X,Y,Z)),exists(K,Z) ⇔ leq(Y,X) |
Z=Y, entailed(K, min(X,Y,Z)).
min(X,Y,M) \ ask(K, min(X,Y,Z)), exists(K,Z) ⇔leq(X,Y) |
Z=M, entailed(K, min(X,Y,Z)).

Each of these rules adds the entailed constraint to the store. This generalizes the solution proposed in [2] for the operational semantics of CHR for built-in constraints, to user-defined constraints.

Syntax and Semantics of CHRat($\mathcal{X}$) Components

Syntax.

In CHRat, control tokens are built with extra symbols with the signature $\Pi_t = \{\text{ask}/2, \text{exists}/2, \text{entailed}/2\}$ disjoint from Π and $\Pi_{\mathcal{X}}$.

Definition 49. A CHRat($\mathcal{X}$) *program* P is a sequence of rules $H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u$ where H is a multiset of elements of $\mathcal{A}$; H' is a multiset of elements of $\mathcal{A} \cup \text{ask}(V, \mathcal{A}) \cup \text{exists}(V, V)$; G is a formula of $\mathcal{L}'$ where $\mathcal{L}'$ is the extension of $\mathcal{L}$ containing atomic propositions $\mathcal{A}$ and closed by conjunction and existential quantification; $B_{\mathcal{X}}$ is a formula of $\mathcal{L}$; and B_u is a multiset of elements of $\mathcal{A} \cup \text{entailed}(V, \mathcal{A})$. H and H' cannot be both empty.

As formula of $\mathcal{L}'$, a guard G is of the form $G_u \wedge G_{\mathcal{X}}$ where G_u is a conjunction (or multiset) of tokens and $G_{\mathcal{X}}$ is a formula of $\mathcal{L}$. The set of variables that only appear in G_u is denoted $V_G = \text{fv}(G_u) \setminus \text{fv}(H, H', G_{\mathcal{X}})$.

Operational Semantics.

Configurations are tuples (T, c, I):

- the built-in store is a constraint $c \in \mathcal{C}$;
- the user store T is a multiset of $\mathcal{A} \cup \text{ask}(V, \mathcal{A}) \cup \text{exists}(V, V) \cup \text{entailed}(V, \mathcal{A})$;
- pending instances table I is a finite support map from variables to rule instances $(\mathbf{r}, \sigma_\pi)$ where $\mathbf{r}$ is a (renamed) rule and σ_π a permutation of $\mathcal{A}$.

For any pending instances table I, $\text{sup}(I)$ denotes the support of I. If $K \in V$ is such that $K \notin \text{sup}(I)$, then $I \uplus (\mathbf{r}, \sigma_\pi)_K$ denotes the map I' such that $\text{sup}(I') = \text{sup}(I) \cup \{K\}$ with $I'|_{\text{sup}(I)} = I$ and $I'(K) = (\mathbf{r}, \sigma_\pi)$.

Two kind of transitions can occur from (T, c, I): suspend $\xrightarrow{\text{s}}$ and wake $\xrightarrow{\text{w}}$.

$$(T, c, I) \xrightarrow{\text{s}} (\langle T \oplus \text{ask}(K, G_u) \oplus \text{exists}(K, V_G)\rangle, \langle c \wedge \exists(c_m)\rangle, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K\rangle)$$

where $\mathbf{r} = \langle H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u \rangle$ is a rule in P renamed with fresh variables with respect to (T, c, I) and π an injection from $T' = H \oplus H'$ to T such that the matching constraint $c_m = \langle (\bigwedge_{t \in \sup(T')} t = \sigma_\pi(t)) \wedge G_{\mathcal{X}} \rangle$ is entailed, i.e. $\mathcal{X} \models \forall \boldsymbol{x}(c \to \exists \boldsymbol{y}(c_m))$ where $\boldsymbol{x} = \mathrm{fv}(c)$ and $\boldsymbol{y} = \mathrm{fv}(c_m) \setminus \mathrm{fv}(c)$. Then the reduction adds the new pending instance $(\mathbf{r}, \sigma_\pi)$ to I, indexed by a fresh variable K.

$$(T, c, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K \rangle) \xrightarrow{\mathrm{w}} (\langle T \ominus \pi'(H') \ominus \pi'(\mathrm{entailed}(K, G_u)) \oplus B_u \rangle, \langle c \wedge c_m \wedge B_{\mathcal{X}} \rangle, I)$$

where $(\mathbf{r}, \sigma_\pi)$ is a pending instance of the rule $\mathbf{r} = \langle H \backslash H' \Rightarrow G \mid B_{\mathcal{X}} \wedge B_u \rangle$ and σ_π represents an injection π from $H \oplus H'$ to T, such that there exists an injection π' from $T' = H \oplus H' \oplus \mathrm{entailed}(K, G_u)$ to T, which extends π such that the matching constraint: $c_m = \langle (\bigwedge_{t \in \sup(T')} t = \sigma_{\pi'}(t)) \wedge G_{\mathcal{X}} \rangle$ is entailed, i.e. $\mathcal{X} \models \forall \boldsymbol{x}(c \to \exists \boldsymbol{y}(c_m))$ where $\boldsymbol{x} = \mathrm{fv}(c)$ and $\boldsymbol{y} = \mathrm{fv}(c_m) \setminus \mathrm{fv}(c)$.

We define $\to = \xrightarrow{\mathrm{s}} \cup \xrightarrow{\mathrm{w}}$.

Definition 50. The *operational semantics* for the observation of accessible and terminal configurations for a query $q = T_0 \wedge c_0 \in \mathcal{L}'$ are, with $\boldsymbol{x} = \mathrm{fv}(T_1, c) \setminus \mathrm{fv}(q)$:

$$\mathcal{O}^a_P[q] = \{\exists \boldsymbol{x}(T_1 \wedge c) \mid (T_0, c_0, \varnothing) \xrightarrow{*} (T, c, \_), T_1 = T \cap \mathcal{A}\}$$
$$\mathcal{O}^t_P[q] = \{\exists \boldsymbol{x}(T_1 \wedge c) \mid (T_0, c_0, \varnothing) \xrightarrow{*} (T, c, \_) \not\to, T_1 = T \cap \mathcal{A}\}$$

The accessible store semantics $\mathcal{O}^a_P[q]$ is related to the logical semantics in the next section. The terminal stores semantics $\mathcal{O}^t_P[q]$ captures the resulting stores once no more rules can be applied. Observed stores are restricted to $\mathcal{A}$. Let min be the solver defined in example 24, then $\mathcal{O}^t_{\min}[\min(X, Y, Z), \mathrm{leq}(X, Y)] = \{\mathrm{leq}(X, Y) \wedge Z = X\}$ and the token $\mathrm{ask}(\mathrm{leq}(Y, X))$ introduced in the unsuccessful guard checking (for the second rule of the leq component) is not exposed.

Logical Semantics.

We define a transformation $(\cdot)^*$ from control tokens and suspended instances to atomic propositions:

$$(\mathrm{ask}(K, p(\overrightarrow{X})))^* = \mathrm{ask}_p(K, \overrightarrow{X})$$
$$(\mathrm{entailed}(K, p(\overrightarrow{X})))^* = \mathrm{entailed}_p(K, \overrightarrow{X})$$
$$(\mathrm{exists}(K, V))^* = \mathrm{exists}(K, V)$$
$$(((H \backslash H' \Rightarrow G \mid B), \sigma_\pi)_K)^* = \mathrm{id}_{\mathbf{r}}(K, \overrightarrow{V}(\sigma_\pi(H \oplus H')))$$

In classical logic, a multiset M of constraints is interpreted by the conjunction $M^\ddagger = \langle \bigwedge_{c \in \sup(M)} c^* \rangle$ of its elements. A rule is logically interpreted as:

$$\left(H \backslash H' \Rightarrow G \;\middle|\; B_{\mathcal{X}}, B_u\right)^\ddagger = \left(H \backslash H' \Rightarrow G \;\middle|\; B_{\mathcal{X}}, B_u\right)^\ddagger_{\mathrm{s}} \wedge \left(H \backslash H' \Rightarrow G \;\middle|\; B_{\mathcal{X}}, B_u\right)^\ddagger_{\mathrm{w}}$$

where sub-formulae suspend and wake translate their operational counterpart:

$$\begin{aligned}(H\backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u)^{\ddagger}_{\mathrm{s}} &= \Big\langle \forall \boldsymbol{x}\Big(\exists \boldsymbol{y}(G) \wedge H^{\ddagger} \wedge H'^{\ddagger} \rightarrow \\ &\qquad \exists K\big(\mathrm{id}(K,\boldsymbol{z}) \wedge \mathrm{exists}(K, V_G) \wedge (\textstyle\bigwedge_{p(\boldsymbol{u})\in\mathrm{sup}(G_u)} \mathrm{ask}_p(K,\boldsymbol{u}))\big)\Big)\Big\rangle \\ (H\backslash H' \Rightarrow G \mid B_{\mathcal{X}}, \wedge B_u)^{\ddagger}_{\mathrm{w}} &= \big\langle \forall \boldsymbol{x} \forall K\big(\exists \boldsymbol{y}(G) \wedge H^{\ddagger} \rightarrow (H'^{\ddagger} \wedge \mathrm{id}(K,\boldsymbol{z}) \wedge \\ &\qquad (\textstyle\bigwedge_{p(\boldsymbol{u})\in G_u} \mathrm{entailed}_p(K,\boldsymbol{u})) \leftrightarrow \exists \boldsymbol{z}(G \wedge B_{\mathcal{X}} \wedge B_u^{\ddagger}))\big)\big\rangle\end{aligned}$$

and $\boldsymbol{x} = \mathrm{fv}(H, H')$, $\boldsymbol{y} = \mathrm{fv}(G) \setminus \mathrm{fv}(H, H')$ and $\boldsymbol{z} = \mathrm{fv}(G, B_{\mathcal{X}}, B_u) \setminus \mathrm{fv}(H, H')$. The logical interpretation of a program $(P)^{\ddagger}$ is the logical conjunction of the interpretations of the rules of P.

Definition 51. The *classical logical semantics* of a query $q \in \mathcal{L}'$ is:

$$\mathcal{L}_P[q] = \{c \in \mathcal{L}' \mid (P)^{\ddagger} \models_{\mathcal{X}} q \rightarrow c\}$$

Let V be the free variables of the initial query $q = T_0 \wedge c_0 \in \mathcal{L}'$. Configurations are interpreted as $(T, c, I)^{\ddagger} = \langle \exists \boldsymbol{x}(T^{\ddagger} \wedge c \wedge (\bigwedge_{K\in\mathrm{sup}(I)} I(K)^*))\rangle$ where $\boldsymbol{x} = \mathrm{fv}(T, c) \setminus \mathrm{fv}(T_0, c_0)$: the query variables appear in the observables and are therefore left free in the interpretation. This definition is extended for a set of configurations S: $S^{\ddagger} = \{(T, c, I)^{\dagger\dagger} \mid (T, c, I) \in S\}$.

With the same restriction as for CHR, CHRat operational semantics is sound w.r.t. the classical logical semantics. This result follows from the following lemma:

Lemma 6. *If* $c_0 \rightarrow c_1$*, then* $(P)^{\ddagger} \models_{\mathcal{X}} (c_0)^{\ddagger} \rightarrow (c_1)^{\ddagger}$.

Proof. If $c_0 \xrightarrow{\mathrm{s}} c_1$, then, by definition of $\xrightarrow{\mathrm{s}}$, there is a rule $\mathbf{r} = \langle H\backslash H' \Rightarrow G | B_{\mathcal{X}}, B_u\rangle$ in P renamed with fresh variables with respect to $c_0 = (T, c, I)$. Moreover, $c_1 = (\langle T \oplus \mathrm{ask}(K, G_u) \oplus \mathrm{exists}(K, V_G)\rangle, \langle c \wedge \exists(c_m)\rangle, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K\rangle)$ where π is an injection from $T' = H \oplus H'$ to T such that the matching constraint $c_m = \langle(\bigwedge_{t\in\mathrm{sup}(T')} t = \sigma_\pi(t)) \wedge G_{\mathcal{X}}\rangle$ is entailed. By definition of $(\cdot)^{\ddagger}$ for configurations, $(c_0)^{\ddagger} = \langle \exists \boldsymbol{x}(T^{\ddagger} \wedge c \wedge (\bigwedge_{K\in\mathrm{sup}(I)} I(K)^*))\rangle$ and $(c_1)^{\ddagger} = \langle \exists \boldsymbol{x} \exists K(T^{\ddagger} \wedge \mathrm{ask}(K, G_u)^{\ddagger} \wedge \mathrm{exists}(K, V_G)^{\ddagger} \wedge c \wedge (\bigwedge_{K\in\mathrm{sup}(I)} I(K)^*) \wedge (\mathbf{r}, \sigma_\pi)^*_K)\rangle$. If $(c_0)^{\ddagger}$, let $\boldsymbol{x}$ such that $T^{\ddagger} \wedge c \wedge (\bigwedge_{K\in\mathrm{sup}(I)} I(K)^*)$. Admitting $(\mathbf{r})^{\ddagger}_{\mathrm{s}}$, we have $c_m \rightarrow (\mathbf{r}, \sigma_\pi)^*_K \wedge \mathrm{exists}(K, V_G)^{\ddagger} \wedge \mathrm{ask}(K, G_u)^{\ddagger}$. Since $(c_0)^{\ddagger} \rightarrow c_m$, we have $(c_1)^{\ddagger}$.

If $c_0 \xrightarrow{\mathrm{w}} c_1$, then, by definition of $\xrightarrow{\mathrm{w}}$, there is a pending instance of the rule $\mathbf{r} = \langle H\backslash H' \Rightarrow G \mid B_{\mathcal{X}} \wedge B_u\rangle$ with fresh variables with respect to $c_0 = (T, c, I \uplus (\mathbf{r}, \sigma_\pi)_K)$. Moreover $c_1 = (\langle T_1 \oplus B_u\rangle, \langle c \wedge c_m \wedge B_{\mathcal{X}}\rangle, I)$ with $T_1 = T \ominus \pi'(H') \ominus \pi'(\mathrm{entailed}(K, G_u))$ and σ_π represents an injection π from $H \oplus H'$ to T, such that there exists an injection π' from $T' = H \oplus H' \oplus \mathrm{entailed}(K, G_u)$ to T, which extends π such that the matching constraint: $c_m = \langle(\bigwedge_{t\in\mathrm{sup}(T')} t = \sigma_{\pi'}(t)) \wedge G_{\mathcal{X}}\rangle$ is entailed. By definition of $(\cdot)^{\ddagger}$ for configurations, $(c_0)^{\ddagger} = \langle \exists \boldsymbol{x}(T_1^{\ddagger} \wedge \pi'(H')^{\ddagger} \wedge$

$\pi'(\text{entailed}(K, G_u))^\ddagger \wedge c \wedge (\bigwedge_{K \in \text{sup}(I)} I(K)^*) \wedge (\mathbf{r}, \sigma_\pi)^*_K)\rangle$ and $(c_1)^\ddagger = \langle \exists \boldsymbol{x}(T_1^\ddagger \wedge B_u^\ddagger \wedge c \wedge c_m \wedge B_{\mathcal{X}}^\ddagger(\bigwedge_{K \in \text{sup}(I)} I(K)^*))\rangle$. If $(c_0)^\ddagger$, let $\boldsymbol{x}$ such that $T_1^\ddagger \wedge \pi'(H')^\ddagger \wedge \pi'(\text{entailed}(K, G_u))^\ddagger \wedge c \wedge (\bigwedge_{K \in \text{sup}(I)} I(K)^*) \wedge (\mathbf{r}, \sigma_\pi)^*_K$. Admitting $(\mathbf{r})^\ddagger_{\text{w}}$, we have $c_m \to B_u^\ddagger \wedge B_{\mathcal{X}}^\ddagger$. Since $(c_0)^\ddagger \to c_m$, we have $(c_1)^\ddagger$. □

Theorem 8. *For any left-linear* CHRat($\mathcal{X}$) *program* P *and initial query* $q \in \mathcal{L}'$*:*

$$\downarrow \mathcal{O}^a_P[q] \subseteq \mathcal{L}_P[q]$$

Proof. Let $q = T_0 \wedge c_0 \in \mathcal{L}'$. Let $\langle \exists \boldsymbol{x}(T_1 \wedge c)\rangle \in \mathcal{O}^a_P[q]$. Then there exists I such that $(T_0, c_0, \varnothing) \xrightarrow{*} (T, c, I)$. Logical implication being transitive, lemma 6 ensures that $(P)^\ddagger \models_{\mathcal{X}} (T_0, c_0, \varnothing)^\ddagger \to (T, c, I)^\ddagger$. Therefore, by definition of $(T_0, c_0, \varnothing)^\ddagger$, $(P)^\ddagger \models_{\mathcal{X}} q \leftrightarrow (T, c, I)^\ddagger$. Therefore, by definition of $\mathcal{L}_P[q]$, $(T, c, I)^\ddagger \in \mathcal{L}_P[q]$. Since $\mathcal{L}_P[q] = \downarrow \mathcal{L}_P[q]$, we have $\downarrow \mathcal{O}^a_P[q] \subseteq \mathcal{L}_P[(q)^\ddagger]$. □

Linear logic semantics use similar atomic propositions to encode control tokens and suspended instances. A multiset M of constraints is interpreted by a linear tensor of the constraints $M^{\ddagger\ddagger} = \langle \bigotimes_{c \in M} c^* \rangle$. A rule is interpreted in linear logic as:

$$\begin{aligned}
\big(H \backslash H' \Rightarrow G \;\big|\; B\big)^{\ddagger\ddagger} &= \big(H \backslash H' \Rightarrow G \;\big|\; B\big)^{\ddagger\ddagger}_{\text{s}} \otimes \big(H \backslash H' \Rightarrow G \;\big|\; B\big)^{\ddagger\ddagger}_{\text{w}} \\
\big(H \backslash H' \Rightarrow G \;\big|\; B_{\mathcal{X}}, B_u\big)^{\ddagger\ddagger}_{\text{s}} &= \Big\langle !\forall \boldsymbol{x}\Big(\exists \boldsymbol{y}(G^{\dagger\dagger}) \otimes H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \multimap \\
&\qquad \exists K\big(H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \otimes \text{id}(K, \boldsymbol{z}) \otimes \text{exists}(K, V_G) \otimes \\
&\qquad (\bigotimes_{p(\boldsymbol{u}) \in G_u} \text{ask}_p(K, \boldsymbol{u}))\big)\Big)\Big\rangle \\
\big(H \backslash H' \Rightarrow G \;\big|\; B_{\mathcal{X}}, B_u\big)^{\ddagger\ddagger}_{\text{w}} &= \Big\langle !\forall \boldsymbol{x} \forall K\Big(\exists \boldsymbol{y}(G^{\dagger\dagger}) \otimes H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \otimes \\
&\qquad \text{id}(K, \boldsymbol{z}) \otimes (\bigotimes_{p(\boldsymbol{u}) \in G_u} \text{entailed}_p(K, \boldsymbol{u})) \multimap \\
&\qquad \exists \boldsymbol{z}(H^{\ddagger\ddagger} G^{\dagger\dagger} \otimes B_{\mathcal{X}} \otimes B_u^{\ddagger\ddagger}))\Big)\Big\rangle
\end{aligned}$$

and $\boldsymbol{x} = \text{fv}(H, H')$, $\boldsymbol{y} = \text{fv}(G) \setminus \text{fv}(H, H')$ and $\boldsymbol{z} = \text{fv}(G, B_{\mathcal{X}}, B_u) \setminus \text{fv}(H, H')$. The logical interpretation of a program $(P)^{\ddagger\ddagger}$ is the linear tenser of the interpretations of the rules of P.

Definition 52. The *linear logical semantics* of a query $q \in \mathcal{L}'$ is:

$$\mathcal{LL}_P[q] = \{c \in \mathcal{L}' \;\big|\; (P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} q \multimap c \otimes \top\}$$

Let V be the free variables of the initial query $q = T_0 \wedge c_0 \in \mathcal{L}'$. Configurations are interpreted as $(T, c, I)^{\ddagger\ddagger} = \langle \exists \boldsymbol{x}(T^{\ddagger\ddagger} \otimes c \otimes (\bigotimes_{K \in I} I(K)^*))\rangle$ where $\boldsymbol{x} = \text{fv}(T, c) \setminus$

$\mathrm{fv}(T_0, c_0)$: the query variables appear in the observables and are therefore left free in the interpretation. This definition is extended for a set of configurations S: $S^{\ddagger\ddagger} = \{(T, c, I)^{\dagger\dagger} \mid (T, c, I) \in S\}$.

CHRat operational semantics is sound and complete w.r.t. the linear logical semantics. This result follows from the following lemma:

Lemma 7. *If* $c_0 \to c_1$*, then* $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} (c_0)^{\ddagger\ddagger} \multimap (c_1)^{\ddagger\ddagger}$.

Proof. If $c_0 \xrightarrow{\mathrm{s}} c_1$, then there is a rule $\mathbf{r} = \langle H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u \rangle$ in P renamed with fresh variables with respect to $c_0 = (T, c, I)$, and $c_1 = (\langle T \oplus \mathrm{ask}(K, G_u) \oplus \mathrm{exists}(K, V_G) \rangle, \langle c \wedge \exists(c_m) \rangle, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K \rangle)$. Then, according to $(\mathbf{r})^{\ddagger\ddagger}_{\mathrm{s}}$, $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} ((T, c, I))^{\ddagger\ddagger} \multimap (c_1)^{\ddagger\ddagger}$.

If $c_0 \xrightarrow{\mathrm{w}} c_1$, then there is a rule $\mathbf{r} = \langle H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u \rangle$ in P renamed with fresh variables with respect to $c_0 = (T, c, I)$, and $c_1 = (\langle T \oplus \mathrm{ask}(K, G_u) \oplus \mathrm{exists}(K, V_G) \rangle, \langle c \wedge \exists(c_m) \rangle, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K \rangle)$. Then, according to $(\mathbf{r})^{\ddagger\ddagger}_{\mathrm{w}}$, $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} ((T, c, I))^{\ddagger\ddagger} \multimap (c_1)^{\ddagger\ddagger}$. □

Soundness and completeness of the operational semantics w.r.t. the linear logical semantics is summarized below:

Theorem 9. *For any* $\mathrm{CHRat}(\mathcal{X})$ *program* P *and initial query* $q \in \mathcal{C}'$*:*

$$\downarrow_{\circ} \mathcal{O}^a_P[q] = \mathcal{LL}_P[q]$$

Proof. Let $q = T_0 \wedge c_0 \in \mathcal{L}'$.

Soundness Let $\langle \exists \boldsymbol{x}(T_1 \wedge c) \rangle \in \mathcal{O}^a_P[q]$. Then there exists I such that $(T_0, c_0, \varnothing) \xrightarrow{*} (T, c, I)$. Logical implication being transitive, lemma 7 ensures that $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} (T_0, c_0, \varnothing)^{\ddagger\ddagger} \multimap (T, c, I)^{\ddagger\ddagger}$. Therefore, by definition of $(T_0, c_0, \varnothing)^{\ddagger\ddagger}$, $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} q \multimap (T, c, I)^{\ddagger\ddagger}$. Therefore, by definition of $\mathcal{LL}_P[q]$, $(T, c, I)^{\ddagger\ddagger} \in \mathcal{LL}_P[q]$. Since $\mathcal{LL}_P[q] = \downarrow_{\circ} \mathcal{LL}_P[q]$, $\downarrow_{\circ} \mathcal{O}^a_P[q] \subseteq \mathcal{LL}_P[(q)^{\ddagger\ddagger}]$.

Completeness *(sketch)* Let $c \in \mathcal{L}_P[(q)^{\ddagger\ddagger}]$. Then $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} q \multimap c$. Main steps of the proof tree are *modus ponens* on linear implications introduced with $(\mathbf{r})^{\ddagger\ddagger}_{\mathrm{s}}$ and $(\mathbf{r})^{\ddagger\ddagger}_{\mathrm{w}}$ which mirror their operational counterpart.

□

Program Transformation from CHRat **to** CHR

Definition 53. Let $[\![\cdot]\!] : \mathrm{CHRat}(\mathcal{X}) \to \mathrm{CHR}(\mathcal{X})$ be the following morphism, where $G_u = \{t_1(\overrightarrow{v_1}), \ldots, t_m(\overrightarrow{v_m})\}$ and $V_G = \{y_1, \ldots, y_p\}$ and $\mathrm{fv}(H, H') = \{x_1, \ldots, x_n\}$:

$$[\![H \backslash H' \Leftrightarrow G | B_{\mathcal{X}}, B_u]\!] = \begin{cases} H^*, H'^* \Rightarrow G_{\mathcal{X}} | \mathrm{id\_}i(K, x_1, \ldots, x_n), \\ \quad \mathrm{exists}(K, y_1), \ldots, \mathrm{exists}(K, y_p), \\ \quad \mathrm{ask\_}t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{ask\_}t_m(K, \overrightarrow{v_m}). \\ H^* \backslash H'^*, \mathrm{id\_}i(K, x_1, \ldots, x_n), \\ \quad \mathrm{entailed\_}t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{entailed\_}t_m(K, \overrightarrow{v_m}) \\ \quad \Rightarrow G_{\mathcal{X}} | B_{\mathcal{X}}, B_u^*. \end{cases}$$

User constraints $\mathrm{ask\_}t_j$, exists, $\mathit{tokentailed\_}t_j$ and $\mathrm{id\_}i$ corresponds to atomic propositions introduced in the logical semantics. i is a unique identifier associated to the rule. The linear logical semantics makes then link between the operational semantics and the transformation defined above: this leads to the following result which proves the soundness of the transformation in the logical semantics:

Theorem 10. *For all* $\mathrm{CHRat}(\mathcal{X})$ *program* P *and initial query* $T_0 \wedge c_0 \in \mathcal{L}'$*:*

$$\mathcal{LL}_P[T_0 \wedge c_0] = \mathcal{LL}_{[\![P]\!]}(T_0, C_0)$$

Proof. Let $H \backslash H' \Leftrightarrow G | B$ be a CHRat rule. $([\![H \backslash H' \Leftrightarrow G | B_{\mathcal{X}}, B_u]\!])^{\dagger\dagger} = S \otimes W$ where:

$$\begin{aligned} S = {} & (H^*, H'^* \Rightarrow G_{\mathcal{X}} | \mathrm{id}(K, x_1, \ldots, x_n), \mathrm{exists}(K, y_1), \ldots, \mathrm{exists}(K, y_p), \\ & \mathrm{ask\_}t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{ask\_}t_m(K, \overrightarrow{v_m}))^{\dagger\dagger} \\ W = {} & (H^* \backslash H'^*, \mathrm{id}(K, x_1, \ldots, x_n), \\ & \mathrm{entailed\_}t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{entailed\_}t_m(K, \overrightarrow{v_m}) \Rightarrow G_{\mathcal{X}} | B_{\mathcal{X}}, B_u^*)^{\dagger\dagger} \end{aligned}$$

By definition, $S = \langle !\forall \boldsymbol{x}(\exists \boldsymbol{y}(G_{\mathcal{X}}^{\dagger\dagger}) \otimes (H^* \oplus H'^*)^{\dagger\dagger} \multimap \exists K((H^* \oplus H'^*)^{\dagger\dagger} \otimes \mathrm{id}(K, \boldsymbol{z}) \otimes \mathrm{exists}(K, V_G) \otimes (\bigotimes_{i=1}^m \mathrm{ask\_}t_i(K, \overrightarrow{v_i})))) \rangle$ where $\boldsymbol{x} = \mathrm{fv}(H, H')$, $\boldsymbol{y} = \mathrm{fv}(G_{\mathcal{X}})$, $\boldsymbol{z} = \{x_1, \ldots, x_n\}$ and $V_G = \{y_1, \ldots, y_p\}$. Since $(H^* \oplus H'^*)^{\dagger\dagger} = H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger}$, $S \equiv ([\![H \backslash H' \Leftrightarrow G | B_{\mathcal{X}}, B_u]\!])_{\mathrm{s}}^{\ddagger\ddagger}$. By definition, $W = \langle !\forall \boldsymbol{x}(\exists \boldsymbol{y}(G_{\mathcal{X}}^{\dagger\dagger}) \otimes H^{\ddagger} \otimes H'^{\ddagger} \otimes \mathrm{id}(K, \boldsymbol{z}) \otimes (\bigotimes_{i=1}^m \mathrm{entailed\_}t_i(K, \overrightarrow{v_i})) \multimap \exists \boldsymbol{u}(H^{\ddagger} \otimes B_{\mathcal{X}}^{\dagger\dagger} \otimes B_u^*)) \rangle$ where $\boldsymbol{u} = \mathrm{fv}(G, B_{\mathcal{X}}, B_u) \setminus \mathrm{fv}(H, H')$. Therefore $W \equiv ([\![H \backslash H' \Leftrightarrow G | B_{\mathcal{X}}, B_u]\!])_{\mathrm{w}}^{\ddagger\ddagger}$. □

4.3.5 Hierarchical Definition of a Rational Terms Constraint Solver

Union-Find Constraint Component

The classical union-find (or disjoint set union) algorithm [77] has been implemented in CHR in [68] with its best-known quasi-linear algorithmic complexity . This positive result is remarkable because declarative languages such as logic programming do not achieve this efficiency [32]. The union-find algorithm maintains a partition of a universe, such that each equivalence class has a *representative* element. Three operations define this data structure:

- make(X) adds the element X to the universe, initially in an equivalence class reduced to the singleton {X}.
- find(X) returns the representative of the equivalence class of X.
- union(X,Y) joins the equivalence classes of X and Y (possibly changing the representative).

As a constraint solver in classical logic, such an algorithm solves the equivalence constraint A $=^{\sim}$ B. The union and find constraint tokens reflect the imperative interpretation of the disjoint-set data structure, which makes representative elements explicit. However, being a representative element is a non-monotonic property which cannot be captured by constraints in classical logic.

Naive Implementation in CHRat.

A first implementation relies on the classical representation of equivalence classes by rooted trees [68]. Roots are representative elements, they are marked as such with the CHR-constraint root(X). Tree branches are marked with A $^{\sim}>$ B, where A is the child and B the parent node.

```
component naive_union_find_solver.
export make/1, =~/2.
make(A) ⇔ root(A).
union(A, B) ⇔ find(A, X), find(B, Y), link(X, Y).
A ~> B \ find(A, X) ⇔ find(B, X).
root(A) \ find(A, X) ⇔ X = A.
link(A, A) ⇔ true.
link(A, B), root(A), root(B) ⇔
        B ~> A, root(A).
```

This implementation supposes that the entry-points make and union are used with constant arguments only, and that the first argument of find is always a constant. Telling this constraint yields to the union of both equivalence classes:

$$\mathrm{A} =^{\sim} \mathrm{B} \Rightarrow \mathrm{union}(\mathrm{A}, \mathrm{B}).$$

A way to provide a naive implementation for the entailment solver of $=^{\sim}$ is to follow the branches until possibly finding a common ancestor for A and B.

$$\mathrm{ask}(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{A}) \Leftrightarrow \mathrm{entailed}\,(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{A}).$$
$$\mathrm{A} \,{}^{\sim}\!> \mathrm{C} \setminus \mathrm{ask}(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{B}) \Leftrightarrow \mathrm{C} =^{\sim} \mathrm{B} \mid \mathrm{entailed}(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{B}).$$
$$\mathrm{B} \,{}^{\sim}\!> \mathrm{C} \setminus \mathrm{ask}(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{B}) \Leftrightarrow \mathrm{A} =^{\sim} \mathrm{C} \mid \mathrm{entailed}(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{B}).$$

The computation required to check the constraint entailment is done by using recursively guards $\mathrm{C} =^{\sim} \mathrm{B}$ and $\mathrm{A} =^{\sim} \mathrm{C}$. Let S be a set of ground elements of $\mathcal{X}$.

Definition 54. A CHR store c *has a valid union-find data-structure* if $=^{\sim}/2$ and root/1 form a forest with elements in S.

Proposition 6. *The property of having a valid union-find data-structure is preserved by all the rules of the solver.*

Proposition 7. *For every* CHR *store c with a valid union-find data-structure and for all elements A and B, A and B share the same equivalence class if and only if, for any fresh variable K, entailed (K, $A =^{\sim} B$) $\in \mathcal{O}_a(ask(K, A =^{\sim} B) \wedge c)$.*

Proof. Suppose that A and B share the same equivalence class and R be the root of the tree representing this class. Let d_1 and d_2 be the length of the $\rightsquigarrow$-path between A and R, and between B and R respectively. Then, the property can be proved by recurrence over $d_1 + d_2$. If $d_1 + d_2 = 0$ then $\mathrm{A} = \mathrm{B} = \mathrm{R}$ and askEq concludes. If $d_1 + d_2 \geqslant 1$ then at least one head of the rules askLeft and askRight matches the store. Suppose without loss of generality that askLeft is applicable. Then there is a $\rightsquigarrow$-path from C to R of length $d_1 - 1$. Therefore the hypothesis induction is applicable and the entailment test of $\mathrm{C} =^{\sim} \mathrm{B}$ succeeds. Then askLeft is fired and entailed (K) is reachable.

Conversely, we first remark that if askEq is never fired, entailed (K) is not reachable. Then, if A and B are not in the same equivalence class, there is no common element between $\rightsquigarrow$-paths starting from A and $\rightsquigarrow$-paths starting from B. By induction on the length of the derivation, every token $\mathrm{ask}(\mathrm{K}, \mathrm{C} =^{\sim} \mathrm{D}) \in \mathcal{O}_a(\mathrm{ask}(\mathrm{K}, \mathrm{A} =^{\sim} \mathrm{B}) \wedge c)$ is such that there is a path between A and C and a path between B and D. Therefore, the rule askEq is never fired. □

Optimized Implementation in CHRat**.**

The second implementation proposed in [68] implements both *path-compression* and *union-by-rank* optimizations to reach the quasi-linear complexity in $\mathrm{O}(n\alpha(n))$.

```
component union_find.
export make/1, =~/2.
make(A) ⇔ root(A, 0).
union(A, B) ⇔ find(A, X), find(B, Y), link(X, Y).
A ~> B, find(A, X) ⇔ find(B, X), A ~> X.
root(A, _) \ find(A, X) ⇔ X = A.
link(A, A) ⇔ true.
link(A, B),root(A, N),root(B, M)⇔N>=M|
  B ~> A, N1 is max(M+1, N), root(A, N1).
link(B, A),root(A, N),root(B, M)⇔N>=M|
  B ~> A, N1 is max(M+1, N), root(A, N1).
```

An optimized check for common equivalence class can rely on find to efficiently get the representatives and then compare them. check(K, A, B, X, Y) represents the knowledge that the equivalence class representatives of A and B are the roots X and Y respectively. When X and Y are known to be equal, entailed(K) is put to the store (checkEq).

```
ask(K, A =~ B) ⇔
        find(A, X), find(B, Y),
        check(K, A, B, X, Y).
root(X) \ check(K, A, B, X, X) ⇔ entailed(K).
```

These two rules are not sufficient to define a complete entailment checker because of the changes applied to the tree structure. Indeed, roots found for A and B can be invalidated by subsequent calls to union, which may transform these roots into child nodes. When a former root becomes a child node, the following two rules put find once again to get the new root.

```
X ~> C \ check(K, A, B, X, Y) ⇔
          find(A, Z), check(K, A, B, Z, Y).
Y ~> C \ check(K, A, B, X, Y) ⇔
          find(B, Z), check(K, A, B, X, Z).
```

Rational Tree Equality Constraint Component

Let us now consider rational terms, i.e. rooted, ordered, unranked, labeled, possibly infinite trees, with a finite number of structurally distinct sub-trees [20]. The nodes are supposed to belong to the universe considered by the union-find solver. Two nodes belonging to the same equivalence class are supposed to be structurally equal. Each node X has a signature F/N, where F is the label of X and N its arity: the associated constraint is denoted fun(X, F, N). For each I between 1 and N, the constraint arg(X, I, Y) states that the Ith sub-tree of X is (structurally equal

to) Y. These constraints have just to be compatible between elements of the same equivalence class:

```
component rational_tree_solver.
import =~/2 from union_find_solver.
export fun/3, arg/3, ~=~/2.
fun(X0, F0, N0) \ fun(X1, F1, N1) ⇔ X0 =~ X1 |
        F0 = F1, N0 = N1.
arg(X0, N, Y0) \ arg(X1, N, Y1) ⇔ X0 =~ X1 | Y0 =~ Y1.
```

Telling that two trees are structurally equal, denoted X ~=~ Y, can be reduced to the union of the two equivalence classes.

```
X ~=~ Y ⇔X =~ Y.
```

The computation associated to asking A ~=~ B requires a co-inductive derivation of structural comparisons to break infinite loops. This is done by memoization: tokens checking(K, A, B) signal that A can be assumed to be equal to B since this check is already in progress. checkTreeAux checks that signatures of A and B are equal and compares arguments.

```
ask(K, A ~=~ B) ⇔checkTree(K, A, B).
checkTree(K, A, B) ⇔ eqTree(K, A, B) | entailed(K, A ~=~ B).
ask(eqTree(K, A, B)) ⇔
    checking(K, A, B), fun(A, FA, NA), fun(B, FB, NB),
    checkTreeAux(K, A, B, FA, NA, FB, NB).
checkTreeAux(K, A, B, F, N, F, N) ⇔
    askArgs(K, A, B, 1, N), collectArgs(K, A, B, 1, N).
```

askArgs adds every askArg token corresponding to each pair of point-wise subtrees of A and B. askArg answers entailedArg if they match. collectArg ensures every entailedArg token have been put before concluding about the entailment of eqTree(K, A, B). It is very close to the definition of an ask solver, but the guard deals with a variable number of tokens equals to the arity of A and B.

```
askArgs(K, A, B, I, N) ⇔ I ⩽ N |
    arg(A, I, AI), arg(B, I, BI),
    askArg(K, A, B, I, AI, BI),
    J is I + 1, askArgs(K, A, B, J, N).
askArgs(K, A, B, I, N) ⇔ true.
collectArgs(K, A, B, I, N), entailedArg(K, A, B, I) ⇔
    J is I + 1, collectArgs(K, A, B, J, N).
collectArgs(K, A, B, I, N) ⇔ I > N |
    entailed(eqTree(K, A, B)).
```

askArg firstly checks if the equality is memoized, otherwise asks for its check.

```
checking(K, AI, BI) \ askArg(K, A, B, I, AI, BI) ⇔
    entailedArg(K, A, B, I).
askArg(K, A, B, I, AI, BI) ⇔ eqTree(K, AI, BI) |
    entailedArg(K, A, B, I).
```

For the sake of simplicity, this program does not perform garbage collection. In particular memoized tokens checking(K,A,B) are never removed from the store, and disentailed constraints are not eliminated.

4.4 The SiLCC Programming Language

A SiLCC program defines an *LCC agent*. The following agent writes "Hello world!" to standard output.

```
io.std.writeln("Hello world!", _)
```

LCC agents are written with the following grammar. All the other constructions (except the foreign function interface) are just convenient notations for usual programming idioms and can be rewritten with these fundamental constructions.

Terms are constructed with variables and domain symbols: the domain includes integers, floating-point values, character strings (double-quoted), Herbrand terms (with simple quoted functors), and other structures that we will detail later on.

```
exists m (
  m.a('f'(1, 2.5, "example"))
   forall x (m.a(x) -> exists k (io.std.writeln(x, k)))
)
```

Variables are bound with the quantifiers forall and exists with the obvious scope. Other variables are free and denote values defined in other files.

Remark 2. When an existentially quantified variable is only needed once, we can simply note it "_". The previous example is therefore equivalent to the code below.

```
exists m (
  m.a('f'(1, 2.5, "example"))
   forall x (m.a(x) -> io.std.writeln(x, _))
)
```

A theoretical description of the semantics for these fundamental constructions is given in [41]. Intuitively, tells add tokens to the store. Asks wait that some tokens are simultaneously present in the store for removing them and executing the ask body. Transient asks are fired only once, persistent asks are fired as many times as there are matching tokens to remove. The term which precedes the dot before a token identifier is called the module of the token.

4.4.1 Procedures

Since they can be fired several times, persistent asks are typically useful to define procedures, as in the following example (note that string concatenation is just obtained by putting strings one after the others).

```
exists m (
    forall s (
        m.hello(s)
    =>
        io.std.writeln("Hello " s "!", _)
    )
    m.hello("You")
    m.hello("Me")
)
```

Asks without universally quantified variables are written by omitting the forall keyword.

```
exists m (
    (
        m.hello()
    =>
        io.std.writeln("Hello word!", _)
    )
    m.hello()
)
```

There is a convenient notation for persistent asks on single head when all the variables appearing in the arguments of the head are universally quantified (typically, procedure definitions): just write the head followed by the body of the agent surrounded by curly brackets.

```
exists m (
    m.hello(s) {
        io.std.writeln("Hello " s "!", _)
    }
    m.hello("You")
    m.hello("Me")
)
```

Remark 3. When a variable is existentially quantified up to the next closing delimiter (if any, the end of the file otherwise), the quantification may be followed just by a comma and parentheses omitted. The previous example is therefore equivalent to the code below.

```
exists m,
m.hello(s) {
    io.std.writeln("Hello " s "!", _)
}

m.hello("You")
m.hello("Me")
```

Consequently, successive introduction of variables correspond to nested existential quantification and reintroducing a variable masks its previous definition.

```
exists m,
m.hello(s) {
    io.std.writeln("Hello " s "!", _)
}

m.hello("You")

(
  exists m,
  m.hello(s) {
    io.std.writeln("Hello " s ".", _)
  }
  m.hello("Me")
)

m.hello("everybody")

exists m,
m.hello("(will not appear)")
```

4.4.2 Sequencing

When agents are put in parallel, as hello("You") and hello("Me") in the previous example, there is no guarantee that the first is executed before or after the second or, worse, that the two executions are not interleaved. To sequence printings, we should use a variable, say k, passed to the continuation argument of writeln ("...", k): writeln tells k.done() when the printing is finished, allowing us to consume it before pursuing with other printings.

```
exists k,
io.std.writeln("One", k)
```

```
(
    k.done()
->
    exists k',
    io.std.writeln("Two", k')
    (
        k'.done()
    ->
        io.std.writeln("Three", _)
    )
)
```

The notation do { ... } allows sequences to be written more conveniently. The previous example is equivalent to the code below.

```
do {
    io.std.writeln("One")
    io.std.writeln("Two")
    io.std.writeln("Three")
}
```

Elements in sequences may use tokens of the form value(\emph{x}) instead of done() to return values. These values are bound with <− notation.

```
exists m,
forall k (
    m.a(k)
=>
    k.value(1)
)
do {
    x &lt;− m.a()
    io.std.writeln(x)
}
```

Remark 4. This example is expanded to the following code.

```
exists m,
forall k (
    m.a(k)
=>
    k.value(1)
)
exists k,
```

```
m.a(k)
 forall  x (k.value(x) -> io.std.writeln(x, _))
```

4.4.3 Arithmetic and functional notation

We can do some arithmetic by using the standard library module data.arith. The following code prints the product of 123 by 45. Note that io.std.writeln waits that v gets a value before printing it.

```
exists  v,
data.arith.mul(123, 45, v)
io.std.writeln(v, _)
```

The functional notations allows such code to be written more elegantly without having to give a name to the intermediary variable: when a token is written in term position, the token is moved and put in parallel with the rest of the agent, a fresh variable is added as its last argument, and the term where the token was is replaced by this variable. The previous example is equivalent to the code below.

```
io.std.writeln(data.arith.mul(123, 45), _)
```

Moreover, we can write x ∗ y instead of data.arith.mul(x, y), and, more generally, the operators +, −, ∗, /, ^, mod are available as usual notations for arithmetic.

The function definition notation defines procedures with an implicit last argument conventionally called result.

```
exists  m,
function  m.square(x) {
    result  = x ∗ x
}
io.std.writeln(m.square(12), _)
```

When the body of the function is reduced to assign a value to result, we may use the alternative notation function \emph{f}(...) = \emph{value}.

```
exists  m,
function  m.square(x) = x ∗ x
io.std.writeln(m.square(12), _)
```

4.4.4 Conditional

data.arith defines arithmetic comparisons as well. The result of a comparison is represented with a module for either a token true() or a token false().

```
exists m,
m.check(22)
m.check(n) {
    exists k,
    data.arith.le(n, 10, k)
    (
        k.true()
    =>
        io.std.writeln(n.string() " is a number less than or equal to 10.", _)
    )
    (
        k.false()
    =>
        io.std.writeln(n.string() " is a number greater than 10.", _)
    )
}
```

Note that all domain values can be converted to character strings by telling a string() token in their module.

The idiom of the pair of asks on true() and false() on the same module can be written more usually as follows. (Note that x <= y is a notation for data.arith.le(x, y), usual notations for other comparisons are available as well.)

```
exists m,
m.check(22)

m.check(n) {
    if n &lt;= 10 do {
        io.std.writeln(n.string() " is a number less than or equal to 10.")
    }
    else do {
        io.std.writeln(n.string() " is a number greater than 10.")
    }
}
```

Comparisons are allowed in asks as well.

```
exists m,
m.check(22)

forall n (m.check(n) n &lt;= 10 =>
    io.std.writeln(n.string() " is a number less than or equal to 10.", _))
```

```
forall n (m.check(n) n > 10 =>
  io.std.writeln(n.string() " is a number greater than 10.", _))
```

4.4.5 Angelism

Executing an agent can be non-deterministic. Typically, suppose that two asks wait for the same token: when this token is told, there are two possible computations whether the first or the second ask is fired. SiLCC has been designed such that all observable computations are effectively observed. Hence, if each of these two asks leads to an observable behavior (so as to say a side-effect, like printing something), both behaviors will be observed.

```
exists m,
m.token()

m.token() {
  io.std.writeln("First branch.", _)
}

m.token() {
  io.std.writeln("Second branch.", _)
}
```

However, these two branches are two distinct paths in the non-deterministic computation produced by the agent. In particular, they cannot share resources.

```
exists m,
m.token()

m.token() {
  io.std.writeln("First branch.", _)
  m.resource1()
}

m.token() {
  io.std.writeln("Second branch.", _)
  m.resource2()
}

(m.resource1() m.resource2() -> io.std.writeln("Will not be executed!", _))
```

In the previous example, the third ask can only be fired if there exists a computation path where both resource1 and resource2 are simultaneously available,

yet each of these resources is produced by a distinct computation path.

The essential consequence of angelism is the decomposability of asks. For example, forall x (a(x) a(x 1) a(x + 2) => ...)+ is equivalent to forall x (a(x) => a(x 1) -> a(x + 2) -> ...)+: since the second agent suspends if one of the three tokens to be consumed is not available, and since there is no side-effects between the three nested asks, the possible partial consumption is just not observed. More generally, every program is rewritten to a kernel language which is indeed limited to asks on single tokens where all arguments are universally quantified. All other forms of asks are decomposed: for example, forall (a(x, x 1) -> ...)+ is decomposed in forall x y (a(x, y) −> if y = x 1 ...)+.

4.4.6 Modules and the package system

There is always a "current" module, which tokens not prefixed with a dot belong to. This module can always be referred by the keyword this. The notation with locally changes the current module.

```
with io.std do {
    write("10 + 2 = ")
    write(10 + 2)
    writeln("")
}
```

The notation module { ... } is an expression returning a fresh module. This module is the current module of the agent written between the curly brackets.

```
exists m,
m.operate(a, op, b, f) {
    with io.std do {
        write(a.string() " " op " " b.string() " = ")
        write(f.get(a, b))
        writeln("")
    }
}
m.operate(12, "*", 5, module { function get(a, b) = a * b })
```

The agent defined in the file \emph{x}.lcc is read with the free variable \emph{x} as current module. When the free variable \emph{x} is used, the file \emph{x}.lcc is loaded.

If the file example.lcc contains:

```
test() {
  io.std.writeln("This is an example.", _)
}
```

the procedure can be called by telling example.test().

Top-level phrases are read with silcc.top as current module. This module reacts to some directives like trace() for debugging purpose.

Files are sought in the current directory and then in the directories enumerated in the SILCC_PATH environment variable, where directories are separated by colons. This variable should typically contains the path to the standard library. If this variable is not defined, the standard library is heuristically sought in the path ../lib relatively to the location of the "silcc" executable.

The free variable \emph{x}.\emph{y}.\emph{z} designates the file \emph{x}/\emph{y}/\em
To be called this way, the file should begin by package \emph{x}.\emph{y} (in order to ensure that the name of the variable is unique).

4.4.7 Recursion

Here is the classical recursive algorithm for solving the Towers of Hanoi.

```
exists m,

m.Hanoi(n, A, B, C, k) {
  k = do {
    if n = 0 do {
      io.std.writeln("I have nothing to do.")
    }
    else if n = 1 do {
      io.std.writeln("I move a disk from " A " to " C)
    }
    else do {
      m.Hanoi(n − 1, A, C, B)
      io.std.writeln("I move a disk from " A " to " C)
      m.Hanoi(n − 1, B, A, C)
    }
  }
}

m.Hanoi(4, "A", "B", "C", _)
```

When the do { ... } is used in term position, it returns the last continuation if the sequence is not empty. If the sequence is empty, it tells the token done() in a fresh module variable and returns it.

4.4.8 Lists and iterations

Lists of values can be constructed with the notation [\emph{v1}, ..., \emph{vn}]. Lists are constructed recursively from the value [] (which is a notation for data.list.nil()) and head :: tail (which is a notation for data.list.cons(head, tail)) where head is the first element and tail the list of the other elements. The length of a list is measurable with data.list.length. List concatenation can be computed with data.list.concat(l0, l1) or just by putting lists one after the other like in (l0 l1) (which is just a notation for data.term.concat(l0, l1), defined for lists and strings).

Iterations can be written with the for notation. In the following example, we use it on lists.

```
do {
  for i in ["one", "two", "three"] do {
    io.std.writeln(i)
  }
}
```

Remark 5. The previous code is just a notation for the following.

```
do {
  ["one", "two", "three"].iter(
    module {
      forall i v (do(i, v) => io.std.writeln(i, v))
    }
  )
}
```

Hence, all modules which implement asks on the iter token can be iterated with for. There is also a version of the for notation without continuation. For iterating on integers, one can use the notation 1 .. 10 (which is rewritten as data.int.range(1, 10) which returns a module implementing iter).

```
do {
  for _ in 1 .. 10 do {
    io.std.writeln("This will be written ten times.")
  }
}
```

4.4.9 Pattern-matching and terms

Lists can be destructed by pattern-matching: pattern-matching generalizes the if/else by allowing the if branch to introduce existential variables. (Note that

let x = v is just a notation for exists x', x' = v exists x, x = x' with the introduction of the fresh variable x' to avoid capturing x in v).

```
let l = ["a", "b", "c"]
if exists hd tl, l = hd :: tl do {
  io.std.writeln("Head: " hd)
  io.std.writeln("Tail: " tl.string())
}
```

Compound terms are constructed by quoting the functor, possibly followed by arguments between parentheses. Functors are mapped to integers, in particular single-character functors are mapped to Unicode code-point. (Note that pattern-matching can be decomposed in several conjunctions to keep intermediate subterms.)

```
let t = 'a compound term'(1, ['f'('a')])
if exists t' c, t = 'a compound term'(_, [t']) and t' = 'f'(c) do {
  io.std.writeln(t.functor())
  io.std.writeln(t'.functor())
  io.std.write_char(c)
  io.std.write_char('\n')
}
```

Telling functor to a compound term returns its functor as a character string, telling arity returns its arity. t[n] returns the nth argument (in general, e[n] is just a notation for e.get(n)).

Indeed, let is just a particular case of pattern-matching where all free variables are implicitly quantified.

```
let t = 'f'("Hello")
let 'f'(x) = t
do { io.std.writeln(x) }
```

for does pattern-matching as well to filter between enumerated items (note that iterating over a compound term means iterating over its arguments).

```
do {
  for [x] in 'f'(["a"], ["b", "c"], ["d"]) do {
    io.std.writeln(x)
  }
}
```

Remark 6. The previous code is just a notation for the following.

```
do {
  'f'(["a"], ["b", "c"], ["d"]).iter(
```

```
        module {
            forall  i  v  (do(i,  v)  =>
               v  =  (
                   if  exists  x,  i  =  [x]  do  {  io.std.writeln(x)  }
                   else  do  {}
               )
            )
        }
    )
}
```

The <− notation for do sequences allows pattern-matching as well. Note the notation for tuples: in the following, the token value has only one argument (a pair). Tuples are just a notation for terms with the functor ''.

```
exists  m,
forall  k  (
   m.a(k)
=>
   k.value((2,  5))
)
do {
  (x,  y)  &amp;lt;− m.a()
  io.std.writeln(x  +  y)
}
```

Remark 7. This example is expanded to the following code.

```
exists  m,
forall  k  (
   m.a(k)
=>
   k.value((2,  5))
)
exists  k,
m.a(k)
forall  x  y  (k.value((x,  y))  −>  io.std.writeln(x  +  y,  _))
```

4.4.10 References

New mutable references may be defined with data.ref.new(\emph{initial value}). They act as a value but can be modified with do { \emph{x} := \emph{new value} } (x := e is just a notation for x.set(e)).

```
let x = data.ref.new(1)

do {
  io.std.writeln(x * 2)
  x := x + 1
  io.std.writeln(x * 2)
}
```

4.4.11 Records

Records are denoted { \emph{field}:\emph{value}; ...; \emph{field}:\emph{value} }. They are modules where each field is a function returning its value, plus support for enumerating fields and values. We can access to the field f in x with the expression x:f which is a notation for x.f() (this notation is not specific to records). In pattern-matching, all fields must match and there should not be other fields except if ... is used.

```
let x = { a: "hello"; b: 42 }

if exists a, x = { a: a ... } do {
  io.std.writeln(a)
}
```

Remark 8. If the value of a field is ommited, *e.g.* "{ f }", then the field takes the value of the variable with the same name: "{ f: f }". The previous example can then be rewritten as follows.

```
let x = { a: "hello"; b: 42 }

if exists a, x = { a ... } do {
  io.std.writeln(a)
}
```

4.4.12 Hash-tables and argument indexing

The expression data.hashtbl.new(\emph{initial size}) (defined in the standard library of the on-going 0.0.1 version) returns a fresh hash-table which associates a fresh variable to each domain value. The expression \emph{tbl}[\emph{key}] returns the value associated to *key*.

```
let t = data.hashtbl.new(17)
t['f'({ a: 1})] = "example"
```

```
t[{u: 2; v: 3}] = "test"
with io.std do {
  writeln(t['f'({ a: 1})])
  writeln(t[{u: 2; v: 3}])
}
```

Hash-tables are useful to implement indexing on token arguments. Since there is only native indexation on token modules, filtering on arguments in asks can be inefficient. This is a consequence of kernel design: the ask

```
forall i (m.a(i) m.b(i) -> ...)
```

is rewritten in

```
forall i j (m.a(i) m.b(j) -> if i = j { ... })
```

and is therefore fired for every pair of tokens a(i) and b(j). However, an hash-table can be introduced to index the tokens m.b(i) on their argument i: each token m.b(i) is consumed to produce a token m'.b() where m' is indexed on i.

```
exists m,
let b-tbl = data.hashtbl.new(17)
forall i (m.b(i) => b-tbl[i].b())
```

The ask on m.a(i) and m.b(i) can now be written with indexing.

```
forall i (m.a(i) b-tbl[i].b() -> io.std.writeln(i, _))
m.a("Test") m.b("Test")
```

It is worth noticing that since b-tbl[i].b() depends on b(i), consuming one or the other is equivalent. Therefore, angelism ensures that user-indexing on b-tbl[i].b() correctly cohabitates with other asks on b(i).

4.4.13 Asks on variable number of tokens

Let say we want to write an agent with an ask equivalent to, by abuse of notation, forall i (m.a(i) m.a(i 1) ... m.a(2 * i) => ...)+. The basic idea is to iterativaly construct chains m'.chain(i, j) in a local module m' by consuming the tokens m.a(i), ..., m.a(j). Since these chains only exist in computation paths where the corresponding a() tokens have been consumed, consuming a chain is equivalent to consume the individual tokens.

```
exists m,
exists m' (
  forall i (m.a(i) => m'.chain(i, i))
  forall i j (m'.chain(i, j) m.a(j + 1) => m'.chain(i, j + 1))
```

```
  forall i (m'.chain(i, 2 * i) => io.std.writeln(i, _))
)

m.a(7) m.a(8) m.a(9) m.a(10) m.a(11) m.a(12) m.a(13) m.a(14)
```

However, the execution is noticeably slow. The m.a(i) tokens should be indexed.

```
exists m,
exists m' (
  let a-tbl = data.hashtbl.new(17)
  forall i (m.a(i) => a-tbl[i].a())

  forall i (m.a(i) => m'.chain(i, i))
  forall i j (m'.chain(i, j) a-tbl[j + 1].a() => m'.chain(i, j + 1))
  forall i (m'.chain(i, 2 * i) => io.std.writeln(i, _))
)

m.a(7) m.a(8) m.a(9) m.a(10) m.a(11) m.a(12) m.a(13) m.a(14)
```

That's better but it is still slow. The number of chains is quadratic compared to the number of m.a(). Since angelism gives us the possibility to program our own checking procedure, we are now free to optimise it and change the complexity. All chains were constructed whereas the only interesting chains are starting from m.a(i) such that m.a(2 * i) is present. Therefore, the computation of chains could be driven by the presence of the two enclosing tokens m.a(i) and m.a(2 * i).

```
exists m,

let a-tbl = data.hashtbl.new(17)
forall i (m.a(i) => a-tbl[i].a())

forall i (
  m.a(i) a-tbl[2 * i].a()
=>
  exists m',
  m'.chain(i)
  forall j (
    m'.chain(j) a-tbl[j + 1].a()
  =>
    if j + 1 = 2 * i - 1 {
      io.std.writeln(i, _)
```

```
        }
        else {
            m'.chain(j + 1)
        }
    )
)
```

m.a(7) m.a(8) m.a(9) m.a(10) m.a(11) m.a(12) m.a(13) m.a(14)

Definition 55. The *kernel language* is induced by the following grammar.

$$a \coloneqq a \otimes a \mid \forall \boldsymbol{x}(t(\boldsymbol{x}) \multimap a) \mid t(\boldsymbol{x}) \mid \exists x(a)$$

Chapter 5

Conclusion

The programming paradigms that have been considered throughout this thesis share the "programs as models" approach: LLCC and CHR have logical semantics, constraint models have relational interpretations. The observables of interest involve the resolution of a search problem: the execution involves the search for a proof, the search for logical consequences, or the search for solutions for a constraint satisfaction problem.

Execution models are the algorithmic counter-part of these observables of interest: an execution model describes a search strategy for exploring the space of proofs, the space of consequences or the space of assignments. It is worth noticing that each of the three transformations have been guided by lifting programming into modeling: the committed-choice semantics for LLCC modeled in CHR, search strategies modeled as constraint satisfaction problems, and extra-logical control operators modeled as blocked paths in the angelic execution of concurrent programs, faithful to the logical interpretation of agents.

These new high-level compilation schemes open new insights for compiler implementation: constraint solvers could be implemented focusing on the default fixed enumeration strategy and efficient propagators for reified constraints, and there are new challenges for efficient angelic execution for concurrent languages, with the simplest form for asks. From the logician point of view, there is still a connection to be made between angelic execution and the compilation scheme that transforms Horn clauses to search trees by exploring all successful execution paths. The memory shared between execution paths, which allows programs to go "beyond search trees" in ClpZinc and that extends control in SiLCC, opens the hope for deeper logic formalisms that would be able to capture these behaviors logically.

5.1 Committed-Choice Semantics for LLCC through CHR

We have defined compositional translations from CHR to LLCC and from labeled LLCC to CHR and proved that semantics are preserved with strong bisimilarity. Both CHR and LLCC languages are based on the same model of concurrent computation, where agents communicate through a shared constraint store, with a synchronization mechanism based on constraint entailment. This is a generalization of the previous links between CHR and linear logic. As the work for modules in LLCC suggests[41], variables and CHR constraints are expressive enough to embed a form of closures, and thus lead to a simple encoding for the λ-calculus.

Whereas the state during a CHR derivation is entirely determined by the contents of constraint stores, an LLCC configuration contains suspended agents as well. The *ask-lifting* transformation reveals that suspensions can be reified as linear tokens, which in turns become CHR constraints: tokens acting as transient asks are consumed whereas tokens acting as persistent asks are propagated.

Behaviors of programs or agents obtained by translation are precisely related to their antecedents by (strong) bisimulation. To our knowledge, only weak bisimulation results [21] were formulated in the literature for CHR before. To achieve strong bisimulation in our case, we have managed to circumscribe collaterally the non-determinism in the naive operational semantics of CHR and in the operational semantics of LLCC. Transition systems considered here are non-labeled: this was sufficient for semantics preservation and there are good intuitions about the pair of involved firing rules at each step. Formalizing these intuitions by labeling with rule names seems feasible but with low interest. However, labels usually serve to follow messages that an agent either sends or receives. A challenge would be to label $\Rightarrow$-transitions by constraints whereas each single transition consumes some while adding others.

The closure encoding may suggest a new programming style, complementary to the imperative RAM-based style recently described [73]. Optimization of the CHR constraints which reify closures could be explored.

These transformations leads to a straightforward implementation of committed-choice LLCC. However, the moot point is to understand the relevance of CHR refined semantics for the translated LLCC agents: the question of control in LLCC has been tackled in the third part of this thesis, with angelic semantics.

5.2 Constraint Model and Search

We have shown that tree search procedures, such as for instance heuristic labeling, dichotomy, interval-splitting, limited discrepancy search, and dynamic symmetry

breaking during search, can be internalized in a constraint model through reified constraints. On the complex dynamic strategy used for solving Korf's benchmark for square packing, we have shown that the implementation overhead is limited to a factor 2, and can be measured on different CSP solvers without particular support for search. We have also shown with an example that the propagation of search constraints can in fact exhibit an exponential speed up, compared to a classical procedural implementation of the search strategy.

It is worth noting that the conversion of search into constraints opens up a whole field of challenges for CSP solvers with limited built-in search strategies. For instance, Korf's packing problem with the complex strategy of [71] has been proposed to the MiniZinc contest with the best known strategy. Therefore, the two sentences of [71] stating that this packing problem "nicely tests the generality of a search method" and is a "more attractive benchmark for placement problems than the perfect square" can now apply to compare a broad range of CSP solvers.

Furthermore, work on complex problems with dedicated heuristics can now become more independent of a particular solver through the modeling of the search strategy in our approach. We have added the needed indexicals to the FlatZinc parser of some solvers (Choco [23], JaCoP [51], SICStus [1], Gecode [17], ortools [37]) and encourage all solver developers to so in order to tackle these new challenging problems for the MiniZinc community.

Finally, as a perspective for future work, the reification of choice point constraints in our scheme is in principle compatible with lazy clause generation techniques [58] and the learning of nogood by using a SAT solver. Such a combination of modeling search by constraints and learning constraints during search is however quite intriguing and will be the matter of future work.

5.3 Angelic Programming

We have described a new execution model for LLCC and CHR, the angelic semantics, which computes all the reachable configuration from an initial LLCC agent or CHR goal.

We introduced a notion of derivation nets for graphical represention of execution paths. These derivation nets allow the description of sharing strategies which make the angelic semantics tractable in practice. In these settings, we illustrate how angelic semantics can result to a fully declarative language, where control is captured by the logical interpretation. Proposed applications give natural solutions in the angelic execution model to questions which are still open with committed-choice: the existence of CHR meta-interpreters, the redefinition of specific user constraint representations, for indexation in particular, and more generally the interleaving of arbitrary computation between head consumption,

allowing deep guards.

We have shown that by letting the programmer define in CHRat not only satisfiability checks but also entailment checks for constraints, this version of CHR becomes fully modular, i.e. constraints defined in one component can be reused in rules and guards in other components without any restriction. We have shown that some classical examples of constraint solvers defined in CHR could easily be modularized in CHRat and reused for building complex constraint solvers.

Interpreting operational semantics (indifferently CHR or LLCC) as a proof search method in linear logic reveals a parallel between the elimination of solving non-determinism and focalization theory [7] which remains to explore.

This work is a move forward to more declarativity, for reducing the gap between the logical interpretation and the effective implementation of the semantics. We hope for more theoretical development of algorithms taking benefits of the angelic semantics, as well as progress for implementation efficiency.

Bibliography

[1] SICS AB. Sicstus 4.2.3, 2012.

[2] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266, Linz, 1997. Springer-Verlag.

[3] Slim Abdennadher, Thom W. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.

[4] Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Journal of Applied Artificial Intelligence*, 14, 2000.

[5] Slim Abdennadher and Heribert Schütz. CHRv: A flexible query language. In *FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.

[6] Carsten Schürmann Anders Schack-Nielsen. Invited talk: The CHR-Celf connection. In Thom Frühwirth and Tom Schrijvers, editors, *Proceedings of the fifth Constraint Handling Rules Workshop CHR'08*, 2008.

[7] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[8] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999.

[9] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.

[10] Eike Best, Frank S. de Boer, and Catuscia Palamidessi. Concurrent constraint programming with information removal. In *Proceedings of Coordination*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[11] Hariolf Betz. Relating coloured petri nets to constraint handling rules. In *Proceedings of the forth Constraint Handling Rules Workshop CHR'07*, pages 33–47, 2007.

[12] Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In *Proceeding of CP 2005, 11th*, pages 137–151, 2005.

[13] Hariolf Betz, Frank Raiser, and Thom Frühwirth. A complete and terminating execution model for Constraint Handling Rules. *Theory and Practice of Logic Programming*, 10(4-6):597–610, 2010.

[14] K. H. Borgwardt. The average number of pivot steps required by the simplex-method is polynomial. *Mathematical Methods of Operations Research*, 1982.

[15] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[16] A. Colmerauer, Henry Kanoui, Robert Pasero, and Philippe Roussel. Un système de communication en français, rapport préliminaire de fin de contrat iria, groupe intelligence artificielle. Technical report, Technical Report, Faculté des Sciences de Luminy, Université Aix-Marseille II, 1972.

[17] The Gecode Community. Gecode 4.2.1, 2013.

[18] Emmanuel Coquery and François Fages. From typing constraints to typed constraint systems in CHR. In *Proceedings of Third workshop on Rule-based Constraint Reasoning and Programming, associated to CP'01*, November 2001.

[19] Emmanuel Coquery and François Fages. A type system for CHR. In *Recent Advances in Constaints, revised selected papers from CSCLP'05*, number 3978 in Lecture Notes in Artificial Intelligence, pages 100–117. Springer-Verlag, 2006.

[20] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[21] Leslie De Koninck. Logical algorithms meets CHR: A meta-complexity theorem for Constraint Handling Rules with rule priorities. *Theory and Practice of Logic Programming*, 2009.

[22] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *Proceedings of PPDP'07, International Conference on Principles and Practice of Declarative Programming, Wroclaw, Poland*, pages 25–36. ACM Press, 2007.

[23] École des Mines de Nantes. Choco 3, 2014.

[24] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[25] Gregory Duck, Maria Garcia de la Banda, and Peter Stuckey. Compiling ask constraints. In *Proceedings of International Conference on Logic Programming ICLP 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[26] Gregory J. Duck, Peter J. Stuckey, Maria Garcia De La Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *In 20th International Conference on Logic Programming (ICLP'04*, pages 90–104. Springer-Verlag, 2004.

[27] Gregory J. Duck, Peter J. Stuckey, Marìa Garcìa de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of PPDP'03, International Conference on Principles and Practice of Declarative Programming, Uppsala, Sweden*, pages 79–90. ACM Press, 2003.

[28] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, February 2001.

[29] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[30] T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[31] Thom W. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[32] Harald Ganzinger. A new metacomplexity theorem for bottom-up logic programs. In *Proceedings of International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2001.

[33] Ian P. Gent and Barbara Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, pages 599–603, 1999.

[34] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[35] K.P. Girish and Sunil Jacob John. Relations and functions in multiset context. *Information Sciences*, 179(6):758–768, 2009.

[36] Cinzia Giusto, Maurizio Gabbrielli, and Maria Chiara Meo. Expressiveness of multiple heads in CHR. In *SOFSEM '09: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, pages 205–216, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] Google. or-tools, 2014.

[38] Rémy Haemmerlé. *Fermetures et Modules dans les Langages Concurrents avec Contraintes fondés sur la Logique Linéaire*. PhD thesis, Univ. Paris 7. Soutenance le 17 janvier 2008, December 2007.

[39] Rémy Haemmerlé and Hariolf Betz. Verification of constraint handling rules using linear logic phase semantics. In *Proceeding of CHR 2008, the fifth Constraint Handling Rules Workshop*, Report Series 08-10. RICS-Linz, July 2008.

[40] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[41] Rémy Haemmerlé, François Fages, and Sylvain Soliman. Closures and modules within linear logic concurrent constraint programming. In V. Arvind and Sanjiva Prasad, editors, *Proceedings of FSTTCS 2007, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 544–556. Springer-Verlag, 2007.

[42] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence*, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[43] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with ai and or techniques. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *AAAI*, pages 660–664. AAAI Press / The MIT Press, 1988.

[44] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998.

[45] J. Jaffar, M .Maher, K .Marriott, and P. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, October 1998.

[46] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[47] Radha Jagadeesan, Vasant Shanbhogue, and Vijay A. Saraswat. Angelic nondeterminism in concurrent constraint programming. Technical report, Xerox Parc, 1991.

[48] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[49] Richard E. Korf. Optimal rectangle packing: Initial results. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003.

[50] R. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[51] Krzysztof Kuchcinski and Radoslaw Szymanek. Jacop 4.0.0, 2013.

[52] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29 – 127, 1978.

[53] R. J. Lipton. The reachability problem requires exponential space. Technical Report 62, New Haven, Connecticut: Yale University, Department of Computer Science, Research, January 1976.

[54] Michael J. Maher. Logic semantics for a class of committed-choice programs. In *In 4th International Conference on Logic Programming (ICLP'87)*. MIT Press, 1987.

[55] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.

[56] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[57] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.

[58] O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 16(9):357–391, 2009.

[59] Heiko Otzbeyer, , and Er Pretschner. Testing concurrent reactive systems with constraint logic programming. In *In Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming*, 2000.

[60] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer Berlin Heidelberg, 2004.

[61] Gordon Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[62] Reza Rafeh, Kim Marriott, Maria Garcia de la Banda, Nicholas Nethercote, and Mark Wallace. Adding search to zinc. In *CP*, pages 624–629, 2008.

[63] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[64] Vijay Saraswat. A brief introduction to linear concurrent constraint programming. Xerox PARC, 1993.

[65] Vijay A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993.

[66] Anders Schack-Nielsen and Carstem Schürmann. Celf — a logical framework for deductive and concurrent systems (system description). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 320–326, Berlin, Heidelberg, 2008. Springer-Verlag.

[67] T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Fruhwirth. Automatic implication checking for CHR constraint solvers. *Electronic Notes in Theoretical Computer Science*, 147:93–111, January 2006.

[68] Tom Schrijvers and Thom W. Frühwirth. Analysing the CHR implementation of unionfind. In *Proceedings of the 19th Workshop on (Constraint) Logic Programming, WCLP'05*, Ulm, Germany, 2005.

[69] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663, 2009.

[70] Tom Schrijvers, Peter J. Stuckey, and Gregory J. Duck. Abstract interpretation for constraint handling rules. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 218–229, New York, NY, USA, 2005. ACM.

[71] Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *Proceedings of CP'08*, volume 5202 of *LNCS*, pages 52–66. Springer-Verlag, 2008.

[72] Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. *Theory and Practice of Logic Programming*, 10(4-6):433–447, 2010.

[73] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *Proceedings of the second Constraint Handling Rules Workshop, at ICLP'05*, pages 3–17, 2005.

[74] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of constraint handling rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–42, 2009.

[75] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules – a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming*, 2, January 2010.

[76] Martin Sulzmann, Jeremy Wazny, and Peter J. Stuckey. A framework for extended algebraic data types. In *In Proc. of FLOPS'06, volume 3945 of LNCS*, pages 47–64. Springer-Verlag, 2006.

[77] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of ACM*, 31(2):245–281, April 1984.

[78] Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.

[79] David L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1972.

[80] D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, 1983.

[81] David S. Warren. The Andorra principle. In *Presented at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden.*, 1988.