A Linear-Logic Semantics for Constraint Handling Rules With Disjunction

Hariolf Betz

Department of Computer Science, University of Ulm
hariolf.betz@uni-ulm.de

Abstract. We motivate and develop a linear logic declarative semantics for CHR$^\vee$, an extension of the CHR programming language that integrates concurrent committed choice with backtrack search and a predefined underlying constraint handler. We show that our semantics maps each of these aspects of the language to a distinct aspect of linear logic. We show how we can use this semantics to reason about derivations in CHR$^\vee$ and we present strong theorems concerning its soundness and completeness.

1 Introduction

A declarative semantics is a highly desirable property for a programming language. It allows to prove various program properties – foremost correctness –, guarantees platform independence and eliminates various sources of error. Furthermore, declarative programs tend to be shorter and clearer as they contain – in the ideal case – only information about the modeled problem and none about control.
Constraint Handling Rules With Disjunction (CHR$^\vee$)[2] is an extension of Constraint Handling Rules (CHR)[11–14]. CHR$^\vee$ is a multi-paradigm logical programming language that seamlessly integrates a predefined underlying constraint solver with the forward reasoning – as inherited from pure CHR – and backtrack search functionality, which allows for a seamless enbedding of Prolog into CHR$^\vee$. As Abdennadher suggested [3], it is therefore as much interesting as a language to reason about declarative paradigms as it is in its own right as a programming language.
The following simple example program expresses in the first line that a bird might either be an albatross or a penguin. In the second line, an integrity constraint is given, stating that a penguin cannot fly. Obviously, for an input `bird` $\wedge$ `flies`, the result must be `albatross` $\wedge$ `flies`.

```
bird ⇔ albatross ∨ penguin.
penguin, flies ⇔ false.
```

Owing to its predecessors in logic and constraint logic programming, both CHR and CHR$^\vee$ feature a declarative semantics in classical logic. However, we have shown in previous work [6] that the classical declarative semantics of CHR reflects the functionality of the program poorly

for certain classes of programs, namely programs featuring destructive update and/or purposeful non-confluence.
In this paper, we extend this linear logic semantics to CHR$^\vee$. Our first example program would then map to the following logical formula.

$$!\,(bird \multimap albatross \oplus penguin) \otimes\ !\,(penguin \otimes flies \multimap 0)$$

This interpretation is indeed faithful to the operational semantics of CHR$^\vee$ as it logically implies the following formula:

$$bird \otimes flies \multimap albatross \otimes flies$$

A conjunction of `bird` and `flies` can be mapped to a conjunction of `albatross` and `flies`.
The declarative semantics that we propose preserves the clear distinction between don't-care and don't-know nondeterminism in CHR$^\vee$ by mapping it to the dualism of internal and external choice in linear logic. While forward reasoning formalisms have seen semantics in linear logic before – such as the π-calculus [19] and the CC/LCC class of programming languages [10] –, none of those formalisms features a dichotomy between don't-care and don't-know nondeterminism. Therefore, our approach is unique in mapping this dichotomy into the framework of linear logic.
This paper is structured as follows: The following section will provide introductions to to the intuitionistic segment of linear logic as well as to the CHR$^\vee$ programming language. In Sect. 3, we recapitulate at first our linear logic semantics for the segment of pure CHR. Then we develop and define the extension of our semantics to CHR$^\vee$. In Sect. 4 we present two theorems concerning the soundness and completeness of our semantics w.r.t. the abstract operational semantics of CHR$^\vee$. In Sect. 5 we will give an example for the application of our semantics. In Sect. 6 we compare our approach to related works and discuss our conclusions.

2 Preliminaries

In this section we will provide a short introduction to the concepts of both intuitionistic linear logic and the CHR$^\vee$ programming language as far as relevant to this paper.

2.1 Intuitionistic Linear Logic

Linear logic, introduced by Girard in 1987 [15], features a fine distinction between internal and external choice and a faithful embedding of classical logic into linear logic. In this paper, we will limit our focus to a commonly used segment of linear logic, called intuitionistic linear logic (**ILL**). The syntax of intuitionistic linear logic is as follows:

$$L ::= p(\bar{t}) \mid L \multimap L \mid L \otimes L \mid L \& L \mid L \oplus L \mid !L \mid \exists x.L \mid \forall x.L \mid \top \mid 1 \mid 0$$

The tokens of (intuitionistic) linear logic are generally considered to represent resources rather than propositions. This terminology reflects the fact that these tokens may be consumed during the process of reasoning in linear logic, as well as its awareness of multiplicities. Concretely, the atomic formula (A) is *not* equivalent to the formula $(A \otimes A)$, pronounced *"A times A"* (or *"both A and A"* in Wadler's terminology [20]). Rather, the former represents one instance, the latter two instances of a certain resource A.
The $\otimes$ *("times") conjunction* (multiplicative conjunction) is nevertheless close to the intuitive notion that we have of classical conjunction: For any two resources A and B, the conjunction $A \otimes B$ denotes the resource that is available iff both A and B are available.
Linear implication $\multimap$ *("lollipop")* is set apart from classical implication by the fact that the preconditions of a linear implication are consumed during the process of reasoning. A formula of the form $A \multimap B$ (Wadler: *"consume A yielding B"*), expresses the fact that we can substitute one instance of a resource A (that we dispose of) for one instance of a resource B. Note that the formula $(A \multimap B)$ is a resource itself, and as such is also used up when applied.
The *! ("bang") modality* marks stable facts or unlimited resources. Arguably, the most important property of a *banged* resource is that it is *not consumed* in the process of reasoning. Thus, the formula $A \otimes\ !(A \multimap B)$ implies e.g. $B \otimes\ !(A \multimap B)$. Furthermore, multiple instances of *banged* resources are *idempotent* and we can apply weakening to them.
The *constant 1* represents the empty resource and it is consequently the neutral element with respect to the " $\otimes$ " connective.
In our context, we will use the turnstile symbol $\vdash$ as an abbreviation for $\vdash_{ILL}$, which denotes deducability w.r.t. the sequent calculus of intuitionistic linear logic as defined in [15].

Example 1. We model the fact that one cup of coffee is two euros as $!(E \otimes E \multimap C)$. A "bottomless cup" is an offer where an unlimited number of refills is included. If we assume that the exact number of refills is completely arbitrary and it also includes the possibility of not getting coffee at all, we can model it as $!(E \otimes E \multimap !C)$. Finally, we can model the fact that sugar is free as $!(1 \multimap S)$.
Together, the latter two formulae imply e.g. that it is possible to get two cups of coffee with sugar for two euros:

$$!(E \otimes E \multimap !C), !(1 \multimap S) \vdash (E \otimes E \multimap C \otimes C \otimes S \otimes S)$$

Another important aspect of linear logic is its distinction between internal and external choice, i.e. between decisions that can be made during the process of reasoning and decisions of undetermined result (e.g. those enforced by the environment).
In classical logic, this distinction is associated with the duality of classical conjunction and disjunction. Linear logic offers two dedicated connectives that explicitly express modes of choice:
The & *("with") conjunction* (additive conjunction) expresses internal choice. E.g. the formula $A\&B$ (Wadler: "*either A or B*") does imply A or B, both conclusions are correct. However, it does *not* imply $A \otimes B$.

The $\oplus$ *disjunction* expresses *external* choice, i.e. a formula of the form $A \oplus B$ in itself neither implies A alone or B alone. In this respect, it is analogous to classical disjunction.
The *constant* $\top$ *("top")* is the resource that all other resources can be mapped to, i.e. for every A, $(A \multimap \top)$ is a tautology. This property makes $\top$ the neutral element with respect to the & conjunction:
The *constant 0* is reasonably close to falsity in classical logic: It represents failure and it is the resource which yields every other resource. It is the neutral element with respect to $\oplus$.

Example 2. On any given day, it could be either sunny or raining. As we have no influence on the weather, we model this as an external choice: $(S \oplus R)$. On a rainy day, our local café has only seats on the inside: $!(R \multimap I)$. On a sunny day, we have the (internal) choice, to sit either on the inside or on the outside: $!(S \multimap I \& O)$. This implies that on any given day, it is at least possible to sit on the inside of the café:

$$!(R \multimap I), !(S \multimap I \& O) \vdash (S \oplus R) \multimap I$$

We can extend intuitionistic linear logic into a first-order system with the quantifiers $\exists$ and $\forall$. The resulting first-order system allows for a faithful embedding of intuitionistic logic. This is widely considered one of the most important features of linear logic. Figure 1 presents *Girard's Translation*[15] of intuitionistic logic into intuitionistic linear logic.

$$\begin{aligned}
p(\bar{t})^G &::= p(\bar{t}) \\
(A \wedge B)^G &::= A^G \& B^G \\
(A \to B)^G &::= (!A^G) \multimap B^G \\
(A \vee B)^G &::= (!A^G) \oplus (!B^G) \\
(\top)^G &::= \top \\
(\bot)^G &::= 0 \\
(\neg A)^G &::= !A^G \multimap 0 \\
(\forall x.A)^G &::= \forall x.(A^G) \\
(\exists x.A)^G &::= \exists x.!(A^G)
\end{aligned}$$

Fig. 1. Translation from intuitionistic logic into linear logic

The operator $\square^G$ represents translation from intuitionistic into linear logic. $p(\bar{t})$ stands for an atomic proposition. Girard has proven in [15] that an intuitionistic sequent $(\Gamma \vdash_{LJ} \alpha)$ is provable iff $\left(!\Gamma^G \vdash \alpha^G\right)$ is provable in linear logic (where $\vdash_{LJ}$ stands for deducibility w.r.t. Gentzen's system LJ for intuitionistic logic).

2.2 Constraint Handling Rules With Disjunction

Constraint Handling Rules (CHR)[11–14] is a concurrent programming language developed in the 1990s by Frühwirth and originally intended

as a portable language extension for the implementation of constraint solvers, which has also come into use as a stand-alone general purpose concurrent programming language. In 1999, Abdennadher proposed CHR with Disjunction ($\text{CHR}^\vee$)[2], which extends CHR with the possibility to include disjunctions in the rule bodies. This allows for backtracking search and reasoning techniques like abduction in CHR programs. In the following, we will introduce $\text{CHR}^\vee$ as a self-contained language.

The Syntax of $\text{CHR}^\vee$ We distinguish two disjoint sets of atomic constraints, which we call *built-in constraints* and *CHR constraints*. While the behavior of CHR constraints is determined by a $\text{CHR}^\vee$ program, we assume that reasoning over the built-in constraints is performed by a predefined (classic) constraint handler. The set of built-in constraints contains at least the constraints *true*, *false* and $\doteq$ for syntactic equality. A built-in constraint is either an atomic formula of intuitionistic logic or a disjunction thereof. A CHR constraint is a non-empty multiset of atomic formulae. This distinction in the treatment of CHR and built-in constraints emphasizes that a set semantics applies to built-in constraints whereas a multiset semantics applies to CHR constraints.
A *goal* is a multiset, the elements of which are either built-in constraints or atomic CHR constraints or *disjunctions of goals*.
Programs in $\text{CHR}^\vee$ consist of two sorts of guarded rules. Simplification rules express the conditional substitution of a CHR constraint with a certain goal. propagation rules express the addition of a goal under certain conditions without removing anything. The syntax of simplification and propagation rules is $(E \Leftrightarrow C|G)$ and $(E \Rightarrow C|G)$, respectively. The rule head E is a CHR constraint, the guard C is a built-in constraint and the body G is a goal. If the guard equals *true*, it can be omitted.
The syntax of $\text{CHR}^\vee$ is given in Fig. 2.

Built-in constraint:	C,D	$::= \{\top\} \mid \{\bot\} \mid c\,(\bar{t}) \mid C \wedge D$	
CHR constraint:	E,F	$::= \{e\,(\bar{t})\} \mid E \cup F$	
Goal:	G,H	$::= \{C\} \mid E \mid G \cup H \mid \{G \vee H\}$	
Simplification rule:	R	$::= (E \Leftrightarrow C \mid G)$	
Propagation rule:	R	$::= (E \Rightarrow C \mid G)$	
CHR program:	P	$::= R_1, \ldots, R_n$	$n \geq 0$

Fig. 2. The syntax of $\text{CHR}^\vee$

CHR states A CHR state is of the form $\langle G; C\rangle$, where G is a goal and C is a built-in constraint. G is called *goal store* and C is called *constraint store*. Of the two, only the goal store can be arbitrarily manipulated by a CHR program. The constraint store is handled by a predefined constraint

handler according to a constraint theory **CT**. Information can only be added to the constraint store but not removed.
In a pure CHR program, there is exactly one CHR state at each moment of its execution. In CHR$^\vee$, this concept is extended to a disjunction of CHR states. A configuration is a disjunction of CHR states of the form $S_1 \vee S_2 \vee \cdots \vee S_n$, where $S_1, S_2, \ldots S_n$ are CHR states. Each CHR state within a configuration represents an independent branch of a search tree. On execution, a CHR$^\vee$ program is given a single state of the form $\langle G; \top \rangle$, i.e. with an empty constraint store. A state of this form is called an *initial state*.
Our notation for CHR states is summarized in Fig. 3. Note that to simplify notation, we allow a disjunction to be empty (ε). Such an empty disjunction is semantically equivalent to a state $\langle false; false \rangle$.

CHR state:	S	$::= \langle G; C \rangle$
Initial state:	S_0	$::= \langle G; \top \rangle$
Disjunction of states:	$\bar{S}, \bar{T}$	$::= \varepsilon \mid S \mid \left(S \vee \bar{S}\right)$

Fig. 3. Notation for CHR$^\vee$ states

Operational Semantics Even for pure CHR, there are actually several variants of the operational semantics. These variants carry over to CHR$^\vee$, so we have to decide which operational semantics to consider.
The original operational semantics for CHR was published in [11] and is known as the *abstract* semantics of CHR. It is the original and most general operational semantics in that every derivation possible in any of the other semantics is also true in the abstract semantics.
In [1], Abdennadher extended the abstract semantics with a token store for propagation rules in order to avoid trivial non-termination. This was extended to the so-called *refined semantics* of CHR [9], which is closest to the execution strategy used in most implementations of CHR. Examples for other relevant operational semantics include especially the semantics of Probabilistic CHR [13], in which each applicable rule has a (weighted) chance of firing.
As our operational semantics is meant as a framework to reason theoretically over CHR$^\vee$ programs, it appears a matter of course that we chose the abstract semantics as the most general of kind for our considerations.
The transition rules that constitute the operational semantics of CHR are summarized in Fig. 4.
Arguably the most important transition rule, **Simplify** performs a conditional substitution of a CHR constraint in the goal store with a different one.
A CHR$^\vee$ rule $(F \Leftrightarrow D \mid H)$ is applicable if the rule head F matches a CHR constraint E in the goal store. CHR does not use unification (as

Prolog does) but one-sided matching, i.e. the variables in the rule head have to be matched with those in the store, *not* vice versa. Furthermore, the constraint store C of the current program state must imply the rule guard D under the constraint theory CT.
If **Simplify** is applied, the constraint E in goal store is substituted by F, and the variable matching $(E \doteq F)$ as well as the guard D are added to the constraint store.
Propagation differs from simplification in that the matched CHR constraints E are kept in the constraint store rather than being removed. In practical terms, this raises the question of how to avoid trivial nontermination, which has been addressed in [1] and [9]. With respect to the abstract semantics, however, propagation rules can be faithfully reduced to simplification rules by adding a copy of the rule head into the rule body in the source code. Hence, we will consider propagation a special case of simplification in this paper.
The **Solve** transition moves a built-in constraint from the goal store to the constraint store. Optionally, the built-in constraint solver can also apply some simplification to the constraint store. However, this is irrelevant for the declarative semantics and shall be ignored in this paper.
If at least one goal in the goal store contains a disjunction, the **Split** rule is applicable. On splitting, the current state is split into a disjunction of two states, in each of which the goal with the disjunction is substituted with one of its disjoint subgoals.

Simplify	
If	$(F \Leftrightarrow D \mid H)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall(C \rightarrow \exists\bar{x}(F \doteq E \wedge D))$
then	$\bar{S} \vee \langle E \cup G; C\rangle \vee \bar{T} \mapsto \bar{S} \vee \langle H \cup G; (F \doteq E) \wedge D \wedge C\rangle \vee \bar{T}$
Propagate	
If	$(F \Rightarrow D \mid H)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall(C \rightarrow \exists\bar{x}(F \doteq E \wedge D))$
then	$\bar{S} \vee \langle E \cup G; C\rangle \vee \bar{T} \mapsto \bar{S} \vee \langle E \cup H \cup G; (F \doteq E) \wedge D \wedge C\rangle \vee \bar{T}$
Solve	
If	$CT \models (C \wedge D_1) \leftrightarrow D_2$
then	$\bar{S} \vee \langle \{C\} \cup G; D_1\rangle \vee \bar{T} \mapsto \bar{S} \vee \langle G; D_2\rangle \vee \bar{T}$
Split	
	$\bar{S} \vee \langle G \cup \{G_1 \vee G_2\}, C\rangle \vee \bar{T} \mapsto \bar{S} \vee \langle G \cup G_1, C\rangle \vee \langle G \cup G_2, C\rangle \vee \bar{T}$

Fig. 4. $\mathrm{CHR}^{\vee}$ transition rules

3 A Linear Logic Semantics for $\mathrm{CHR}^{\vee}$

In this section, we will at first recapitulate the previously published linear logic semantics for the segment of pure CHR[6] and then discuss how to extend this result to the full segment of $\mathrm{CHR}^{\vee}$.

3.1 The Linear Logic Semantics for pure CHR

Our semantics is based on the observation that the **Simplify** transition of CHR behaves similarly to the modus ponens of linear logic. Both mechanisms can be characterized as the rewriting of one or several logical predicates in a context that behaves as a multiset. Hence, we translate simplifiaction rules to linear implications and, consequently, CHR states to multiplicative conjunctions of linear predicates.
In the general case – and unlike linear implication – simplifiaction rules are guarded. We can straightforwardly translate the guard into another precondition of the corresponding linear implication. We must make sure, however, that the predicates that correspond with the built-in constraints are not "consumed" during the process. Therefore, we introduce the convention that built-in constraints appear only banged throughout our semantics. As a side-effect, this helps to distinguish built-in constraints from CHR constraints.
The built-in constants *true* and *false* represent the empty constraint and failure, which is why we map them to the linear constants 1 and 0, respectively.
A couple of adaptations have to be made to the underlying constraint theory CT according to which the built-in constraints are handled. As Girard showed in [15], it is possible to faithfully translate intuitionistic logic into intuitionistic linear logic (ILL). If we require that the theory CT be a theory of intuitionistic logic, we can thus translate it to ILL.
Secondly, we have to make sure that the equality constraint is handled correctly. For this purpose, we add a number of formulae to the constraint theory. Concretely, for every n-ary CHR constraint E, we add the the following conjunction of formulae to the constraint theory:

$$\bigotimes_{j=1}^{n} !\forall.\, \big(E(t_1,...,t_j,...,t_n) \otimes !(t_j \doteq t'_j) \multimap E(t_1,...,t'_j,...,t_n)\big)$$

E.g. for a CHR constraint `edge/2`, we would add the following formula to the constraint theory:

$$!\forall.\, \big(edge(t_1,t_2) \otimes !(t_1 \doteq t'_1) \multimap edge(t'_1,t_2)\big) \;\otimes$$
$$!\forall.\, \big(edge(t_1,t_2) \otimes !(t_2 \doteq t'_2) \multimap edge(t_1,t'_2)\big)$$

This translated version of the constraint theory CT, including the formulae for equality, will be called *linear constraint theory* LCT. For every intuitionistic formula $CT \vdash_{LJ} \varphi$, we have $!LCT \vdash \varphi^G$ where $\Box^G$ represents translation from intuitionistic logic to linear logic according to [15].

3.2 Extending the Semantics

In this section we will explain the basic idea of our proposed semantics. We will first show how don't-care nondeterminism is already represented in the linear logic semantics of pure CHR. Then we will discuss how we can extend this semantics for $\text{CHR}^\vee$.

In the linear logic semantics for pure CHR, the program is mapped to a $\otimes$ conjunction of formulae of the form $!\forall\left(!D_i^L \multimap F_i^L \multimap \exists \bar{x}_i.H_i^L\right)$.
This translation logically implies *internal* choice whenever more than one rule is applicable. We consider the following program P_1:

$\mathtt{F} \Leftrightarrow \mathtt{D} \mid \mathtt{H}_1$
$\mathtt{F} \Leftrightarrow \mathtt{D} \mid \mathtt{H}_2$

The logical reading of this program is:

$$P_1^L = !\forall\left(!D^L \multimap F^L \multimap \exists \bar{x}_1.H_1^L\right) \otimes !\forall\left(!D^L \multimap F^L \multimap \exists \bar{x}_2.H_2^L\right)$$

This is logically equivalent to:

$$!\forall\left(!D^L \otimes F^L \multimap (\exists \bar{x}_1.H_1^L) \& (\exists \bar{x}_2.H_2^L)\right)$$

Operationally, the case of several applicable rules is handled by *committed choice*, i.e. as don't-care nondeterminism. Thus, the linear logic semantics of CHR contains an *implicit* mapping of don't-care nondeterminism to internal choice.
The syntax of CHR$^\vee$ differs from the syntax of pure CHR only in the presence of disjunctions in goals. Operationally, disjunction is evaluated as don't-know nondeterminism, i.e. when one of several options is chosen, the other options will *not* be discarded. Analogously to Prolog, don't-know nondeterminism is usually implemented as backtracking.
We will consider another example program P_2, with disjunction:

$\mathtt{H} \Leftrightarrow \mathtt{D} \mid \mathtt{F}_1 \ \vee \ \mathtt{F}_2$

We consider the derivation that results from the same program state that we used in our previous example:

$$\langle H; D\rangle \mapsto_{\texttt{Simplify}} \langle F_1 \vee F_2; D\rangle \mapsto_{\texttt{Split}} \langle F_1; D\rangle \vee \langle F_2; D\rangle$$

We notice that there is no *choice* between $\langle F_1; D\rangle$ and $\langle F_2; D\rangle$ here. Instead, there is a disjunction $\langle F_1; D\rangle \vee \langle F_2; D\rangle$ and there is no way to remove any of the two. By contrast, in our previous example program P_1, a state $\langle H; D\rangle$ might be followed by either $\langle F_1; D\rangle$ or $\langle F_2; D\rangle$. This is reflected in its logical reading: Both sequents $P_1^L \vdash \langle H; D\rangle^L \multimap \langle F_1; D\rangle^L$ and $P_1^L \vdash \langle H; D\rangle^L \multimap \langle F_2; D\rangle^L$ are provable.
The logical reading of program P_2 should reflect the fact that we can remove neither $\langle F_1; D\rangle$ nor $\langle F_2; D\rangle$. However, it also has to reflect the fact that $\langle F_1; D\rangle$ and $\langle F_2; D\rangle$ are processed independently from each other.
Therefore, disjunctions of states and disjunctions in goals will be translated to $\oplus$ disjunctions. Under this semantics, example program P_2 translates as follows:

$$P_2^L = !\forall\left(!D^L \otimes H^L \multimap (\exists \bar{x}_1.F_1^L) \oplus (\exists \bar{x}_2.F_2^L)\right)$$

The translation of our disjunction of states will likewise be an $\oplus$ disjunction:

$$\langle F_1; D\rangle \oplus \langle F_2; D\rangle$$

This translation satisfies all of our conditions and the sequent $P_2^L \vdash \langle H; D\rangle^L \multimap \langle F_1; D\rangle^L \oplus \langle F_2; D\rangle^L$ is provable.
Failed states, i.e. states of the form $\langle G; false\rangle$, translate to 0, which is the neutral element w.r.t. $\oplus$. Thus we have also modelled the removal of failed states from the current disjunction of states.

3.3 Definition of the Extended Semantics

Figure 5 presents our linear logic semantics for CHR$^\vee$. The operator $\square^L$ represents translation into linear logic.

Built-in constraints:	$\top^L$	::=	1
	$\bot^L$	::=	0
	$c(\bar{t})^L$	::=	$!c(\bar{t})$
	$(C \wedge D)^L$	::=	$C \otimes D$
CHR constraints:	$\{e(\bar{t})\}^L$	::=	$e(\bar{t})$
	$(E \cup F)^L$	::=	$E^L \otimes F^L$
Goals:	$\{C\}^L$	::=	C^L
	$(G \cup H)^L$	::=	$G^L \otimes H^L$
	$(G \vee H)^L$	::=	$G^L \oplus H^L$
Initial states:	$S_0^L = \langle G; true\rangle^L$	::=	G^L
Derived states:	$S_a^L = \langle G; C\rangle^L$	::=	$\exists \bar{x}_a (G^L \otimes C^L)$
Parallel states:	$(S \vee T)^L$	::=	$S^L \oplus T^L$
Simplification rules:	$(E \Leftrightarrow C \mid G)^L$	::=	$!\forall \left(!C^L \otimes E^L \multimap \exists \bar{y}.G^L\right)$
Propagation rules:	$(E \Rightarrow C \mid G)^L$	::=	$!\forall \left(!C^L \otimes E^L \multimap E^L \otimes \exists \bar{y}.G^L\right)$
Programs:	$(R_1 ... R_m)^L$	::=	$R_1^L \otimes ... \otimes R_m^L$

Fig. 5. Linear-logic declarative semantics

Both types of constraints are mapped to $\otimes$ conjunctions of atomic constraints, atomic built-in constraints are banged. $!C^G$. Goals containing disjunctions and disjunctions of states are mapped to $\oplus$ disjunctions.
Initial states are translated as goals, whereas for a non-initial state S_a, the local variables $\bar{x}_a$ of S_a – i.e. the variables that do not appear in the corresponding initial state S_0 – are existentially quantified.
CHR$^\vee$ rules are mapped to linear implications with the translations of head and guard on the condition side and that of the body as consequence. The local variables $\bar{y}$ of the rule body are existentially quantified. The bang before the translation of the guard is actually redundant but kept for formal clarity. A CHR$^\vee$ program is translated into a "$\otimes$" conjunction of the translation of its rules.

4 Soundness and Completeness

Concerning the soundness and completeness of our declarative semantics w.r.t. the operational semantics, the following theorems hold.

Theorem 1 (Soundness). *If S_0 is an initial CHR state and $\bar{S}_n$ is a disjunction of states, which is derivable from S_0 under a program P and a linear constraint theory **LCT**. Then the following holds:*

$$!LCT, P^L \vdash \forall \left(S_0^L \multimap \bar{S}_n^L \right)$$

Theorem 2 (Completeness). *If S_0 is an initial CHR state and $\bar{S}_n$ is a disjunction of states, such that*

$$!LCT, P^L \vdash \forall \left(S_0^L \multimap \bar{S}_n^L \right)$$

then there is a disjunction of states $\bar{S}_\nu$ with a finite derivation $S_0 \mapsto^* \bar{S}_\nu$ such that

$$!LCT \vdash \bar{S}_\nu^L \multimap \bar{S}_n^L$$

Theorem 2 states that if an initial CHR$^\vee$ state S_i logically implies a disjunction $\bar{S}_n$ of CHR$^\vee$ states then it is operationally possible to reach a disjunction of states $\bar{S}_\nu$ that implies $\bar{S}_n$ under the constraint theory CT.

Proof Sketch While Theorem 1 can be reduced to the proof of the soundness theorem for the linear logic semantics for pure CHR, the proof of Theorem 2 is more challenging. The first major point to show is that the sequent $!LCT, P^L \vdash \forall \left(S_0^L \multimap \bar{S}_n^L \right)$ can be transformed to a logically equivalent sequent which we call *restricted*. In that restricted sequent, every logical reading ($!\rho$) of a CHR$^\vee$ rule is substituted by a finite number of formulae of the form ($1\&\rho$). This means that the number of CHR$^\vee$ rule applications is not strictly determined, but limited. (Strict determination is not always possible.) We prove this by defining a set of transformation rules that transform a formal proof of our original sequent into a formal proof of the restricted sequent.
We then show that we can apply CHR$^\vee$ transformations to the state S_0, which corresponds to S_0^L, such that the logical reading of the resulting state also implies $\bar{S}_n^L$. This is easy to show for **Solve** and **Split**. For the case that none of those are applicable, we show that *either* a formula ($1\&\rho$) in the transformed sequent corresponds to an *applicable* CHR$^\vee$ rule, *or* we have already found the state $\bar{S}_\nu$. We do this by induction over the sequents of a formal proof. The finite number of sub-formulae of the form ($1\&\rho$) implies that we can derive $\bar{S}_\nu$ with a *finite* derivation.

Significance of Theorem 2: The linear constraint theory LCT determines the handling of the built-in constraints only, it does not have an effect on the actual CHR constraints. Consequently, for every CHR state $\langle E_\nu; C_\nu \rangle$ in $\bar{S}_\nu$, there is a CHR state $\langle E_n; C_n \rangle$ in $\bar{S}_n$ such that E_ν and E_n differ *only* in the built-in constraints. Another notable point is that the derivation $S_0 \mapsto^* \bar{S}_\nu$ is explicitly finite.
Furthermore, our semantics implicitly defines an interesting segment of intuitionistic linear logic, consisting of all sequents of the form $!LCT, P^L \vdash \forall \left(S_0^L \multimap \bar{S}_n^L \right)$. For any such sequent, it is a necessary condition for its provability that a finite number of *modus ponens* applications

– mimicking **Simplify** transitions – can simplify it to a sequent of the form $!LCT \vdash \bar{S}_\nu^L \multimap \bar{S}_n^L$ where the proof of the latter sequent can be reduced to a proof in classical intuitionistic logic. Furthermore, if the *modus ponens* applications and a proof for $!LCT \vdash \bar{S}_\nu^L \multimap \bar{S}_n^L$ are known, these can be used to construct a proof for $!LCT, P^L \vdash \forall\left(S_0^L \multimap \bar{S}_n^L\right)$. We suppose that these findings will enable us to make proof search in our specific segment of linear logic significantly more efficient.

5 Example

This example will show how we can apply our linear logic semantics to reason about CHR$^\vee$ programs that integrate several programming paradigms. The following classic Prolog program implements a ternary append predicate for lists, representing the fact that the third argument is a concatenation of the first two.

```
append(X,Y,Z) ← X≐[], Y≐L, Z≐L.
append(X,Y,Z) ← X≐[H|L1], Y≐L2, Z≐[H|L3],append(L1,L2,L3).
```

We can embed this program faithfully into CHR$^\vee$ by explicitly stating the don't-know nondeterminism using the $\vee$ operator.

```
app1@ append(X,Y,Z) ⇔
    ( X≐[], Y≐L, Z≐L
    ∨ X≐[H|L1], Y≐L2, Z≐[H|L3], append(L1,L2,L3) ).
```

The linear logic reading of this program looks as follows:

$$!\forall X, Y, Z.\,(\mathit{append}(X, Y, Z) \multimap \exists L, L1, L2, L3, H.\\ (!(X \doteq []) \otimes !(Y \doteq L) \otimes !(Z \doteq L)) \oplus\\ (!(X \doteq [H|L1]) \otimes !(Y \doteq L2) \otimes !(Z \doteq [H|L3]) \otimes \mathit{append}(L1, L2, L3)))$$

Now we want to add a rule that should not change the overall semantics of the program but increase efficiency by intercepting the worst case, where the second argument is an empty list. Note that the introduction of this rule adds don't-care nondeterminism to the program. The rule looks as follows:

```
app2@ append(X,Y,Z) ⇔ Y≐[] | X≐Z.
```

The rule `app2` corresponds to the following logical reading:

$$!\forall X, Y, Z.\,(!(Y \doteq []) \otimes \mathit{append}(X, Y, Z) \multimap X \doteq Z)$$

By induction over the length of the list `X` – i.e. the second argument of `append(X,Y,Z)` – we can show that the logical reading of `app2` is entailed by the logical reading of `app1`. Hence, the program consisting of rule `app1` only and the program consisting of both `app1` and `app2` are operationally equivalent.

6 Discussion

6.1 Related Work

Common linear logic languages such as LO[4], Lygon[16] and Lolli[18] rely on a backchaining operationalization of linear logic. Thus, they are not directly comparably to our linear logic semantics of an existing forward-chaining programming language.
The most closely related approaches to this work are therefore Miller's linear logic semantics for the π-calculus [19] and the linear logic semantics for the CC/LCC class of programming languages by Fages, Ruet and Soliman [10]. In Miller's approach, it is don't-care – not don't-know – nondeterminism that is being mapped to linear additive disjunction $\oplus$. Fages et al are closer to our approach in that they explicitly map don't-care nondeterminism to additive conjunction (which we do implicitly). However, neither the π-calculus nor the CC/LCC class of languages feature the dichotomy between don't-know and don't-care nondeterminism that CHRv does.
Djelloul, Meister and Robin have recently related transaction logic to CHR in [8]. The main difference here is that their approach covers only the so-called *range-restricted ground* segment of CHR whereas our approach covers the full segment of $\text{CHR}^\vee$.
If not directly related to our work, we find it worth to mention two approaches to a logical characterization of the dichotomy of forward and backward chaining: In [17], Harland, Pym, and Winikoff incorporate forward reasoning features directly into the sequent calculus of linear logic. In [7], Chaudhuri, Pfenning, and Price improved the focusing inverse method for linear logic such that it generalizes both forward-chaining and backward-chaining proof search. These works are loosely related to our work because the don't-know nondeterminism of $\text{CHR}^\vee$ can be used to embed SLD resolution, which is inherently a backward chaining approach.

6.2 Conclusion

We have presented a linear logic semantics for $\text{CHR}^\vee$. The core of our result is the mapping of the dualism between don't-care and don't-know nondeterminism in $\text{CHR}^\vee$ to the dualism of internal and external choice in linear logic. Furthermore, we use Girard's translation to embed the constraint theory CT that is handled by the built-in constraint handlers. This semantics provides us with a powerful formalism to reason about $\text{CHR}^\vee$ programs.
As $\text{CHR}^\vee$ is considered to be a formalism to experiment with logical programming paradigms, we expect to be able to apply our result to other programming languages and paradigms that mix classical constraint handling, forward chaining and backward chaining.
The applicability of our proposed semantics and its model checking capabilities offers a promising field for future research. Another aspect for future work would would be the design of an algorithm to efficiently find linear logic proofs in the segment of linear logic that is defined by our

declarative semantics. Obviously, this segment is much smaller than the full segment of intuitionistic linear logic and therefore the efficiency of proof search might be increased significantly. Furthermore, our completeness theorem and its proof seem to suggest approaches on how to limit the search space significantly.

Acknowledgements The author wishes to express his gratitude to the reviewers of an earlier version of this paper for their helpful remarks, to his supervisor Thom Frühwirth and to the University of Ulm for funding him with LGFG grant #0518.

References

1. S. Abdennadher: *Operational Semantics and Confluence of Constraint Handling Rules*. Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP 1997), Austria, October 1997.
2. S. Abdennadher, H. Schütz: *CHRv: A Flexible Query Language*. International conference on Flexible Query Answering Systems, FQAS'98, Springer LNCS, Roskilde, Denmark, May 1998
3. S. Abdennadher: *A Language for Experimenting with Declarative Paradigms*. Second Workshop on Rule-Based Constraint Reasoning and Programming, Singapore, 2000.
4. J. Andreoli and R. Pareschi: *LO and Behold! Concurrent Structured Processes*. ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, 25(10):44-56, 1990.
5. H. Betz: *A Linear Logic Semantics for CHR*. Master Thesis, University of Ulm, October 2004, [www.informatik.uni-ulm.de/pm/mitarbeiter/ fruehwirth/other/betzdipl.ps.gz].
6. H. Betz, T. Frühwirth: *A Linear-Logic Semantics For Constraint Handling Rules*. Proceedings of CP 2005, 137-151, Springer, 2005.
7. K. Chaudhuri, F. Pfenning, G. Price: *A logical characterization of forward and backward chaining in the inverse method.* Proceedings of IJCAR'06, pp. 97-111, Seattle, Springer LNCS 4130, August 2006.
8. K. Djelloul, M. Meister, J. Robin. *A Unified Semantics for Constraint Handling Rules in Transaction Logic*. Proceedings of LPNMR'2007, Tempe, AZ, USA, 2007.
9. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur: *The Refined Operational Semantics of Constraint Handling Rules*. Proceedings of the 20th International Conference on Logic Programming, 2004.
10. F. Fages, P. Ruet, S. Soliman: *Linear concurrent constraint programming: linear and phase semantics*. Information and Computation, 165(1):14-41, 2001.
11. T. Frühwirth: *Constraint Handling Rules*. Constraint Programming: Basics and Trends, Springer LNCS 910, 1995.
12. T. Frühwirth: *Theory and Practice of Constraint Handling Rules*, Journal of Logic Programming, 37(1-3):95-138, 1998.

13. T. Frühwirth, A. Di Pierro, and H. Wiklicky: *Probabilistic Constraint Handling Rules*, 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002), 2002.
14. T. Frühwirth, S. Abdennadher: *Essentials of Constraint Programming*, Springer, 2003.
15. J.-Y. Girard: *Linear Logic: Its syntax and semantics*. Theoretical Computer Science, 50:1-102, 1987.
16. J. Harland, D. Pym, and M. Winikoff: *Programming in Lygon: An Overview*. Springer LNCS 1101:391-405, Munich, 1996.
17. J. Harland, D. Pym, and M. Winikoff: *Forward and Backward Chaining in Linear Logic*. Proceedings of the CADE-17 Workhop on Proof-Search in Type-Theoretic Systems, Pittsburgh, 2000.
18. J. Hodas, D. Miller: *Logic Programming in a Fragment of Intuitionistic Linear Logic*. Proceedings 6th IEEE Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, New York, 1991.
19. D. Miller: *The π-calculus as a theory in linear logic: Preliminary results*. ELP 1992: 242-264, Bologna, Italy, 1992.
20. P. Wadler: *A Taste of Linear Logic*. Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk, 1993.