A Linear-Logic Semantics for Constraint Handling Rules

Hariolf Betz and Thom Frühwirth

Faculty of Computer Science, University of Ulm, Germany

Abstract. One of the attractive features of the Constraint Handling Rules (CHR) programming language is its declarative semantics where rules are read as formulae in first-order predicate logic. However, the more CHR is used as a general-purpose programming language, the more the limitations of that kind of declarative semantics in modelling change become apparent. We propose an alternative declarative semantics based on (intuitionistic) linear logic, establishing strong theorems on both soundness and completeness of the new declarative semantics w.r.t. operational semantics.

1 Introduction

Constraint Handling Rules (CHR) is a concurrent committed-choice constraint logic programming language, which was developed in the 1990s as an enhancement to the constraint programming paradigm. Its aim was to add flexibility and customizability to constraint programming by allowing for user-defined constraint-handlers. This is achieved by implementation of the eponymic *constraint handling rules*, which define the rewriting and transformation of conjunctions of atomic formulae.

However, over time CHR has proven useful for many tasks outside its original field of application in constraint reasoning and computational logic, be it agent programming, multi-set rewriting or production rules.

Owing to the tradition of logic and constraint logic programming, CHR features – besides a well-defined operational semantics, of course – a *declarative semantics*, i.e. a direct translation of a CHR program into a first-order logical formula. In the case of constraint handlers, this is a useful tool, since it strongly facilitates proofs of a program's faithful handling of constraints.

The classical-logic declarative semantics, however, poses a problem, when applied to non-traditional uses of CHR, i.e. CHR programs that use CHR as a general-purpose concurrent programming language. Many implemented algorithms do not have a first-order classical logical reading, especially when these algorithms are deliberately non-confluent[1]. This may lead to logical readings which are inconsistent with the intended meaning. This problem has recently been demonstrated in [9] and constitutes the motivation for our development of an alternative declarative semantics.

[1] Meaning that different rule applications may lead to different final results.

Example 1. For an example of an inconsistent classical reading, consider the following coin-throw simulator.

$$\texttt{throw(Coin)} \Leftrightarrow \texttt{Coin} = \texttt{head} \text{ (r1)}$$
$$\texttt{throw(Coin)} \Leftrightarrow \texttt{Coin} = \texttt{tail} \text{ (r2)}$$

The program handles the constraint *throw(Coin)* by committing to one of the rules, thereby equating either *head* or *tail* with the variable *Coin*. (This requires a fair selection strategy.)

Its classical declarative semantics is:

$$(throw(Coin) \leftrightarrow Coin = head) \wedge (throw(Coin) \leftrightarrow Coin = tail)$$

From this we would conclude $(Coin = head) \leftrightarrow (Coin = tail)$ and therefore $head = tail$. In natural language: Both sides of our coin are equal.

Obviously, this statement is not consistent with the intuitive idea of a coin throw. What our program describes is an algorithm, respectively a course of action. The logical reading misinterprets it as a description of stable facts. This shows the basic incompatibility between the classical declarative semantics and non-traditional CHR programs. (Non-traditional in the sense that it is not only a constraint handler.) First-order logic can in general not handle updates, change, dynamics without resorting to auxiliary constructions like adding time-stamps to predicates.

With the linear-logic declarative semantics as we will propose in Sect. 4 we get the following logical reading:

$$!\,(throw(Coin) \multimap (Coin = head) \& (Coin = tail))$$

Informally speaking, the above expression of linear logic says that we can replace $throw(Coin)$ with either $(Coin = head)$ or $(Coin = tail)$ but not both, i.e. a committed choice takes place.

Ever since its introduction in 1987, linear logic has inspired uses as a means to logically formalize computational structures and dynamics. Several programming languages have been designed from scratch for the purpose of making linear logic executable. E.g. the programming language Linear Objects (LO) [3] extends Horn logic by an "additive" conjunction (as occurs in linear logic) to model structured processes. A more formal approach is taken with Lygon [11]. Lygon is based on a systematic proof-theoretic analysis of linear logic, which results in a large segment of linear logic to be covered.

As we will see, there are remarkable similarities between linear logic and the operational semantics of Constraint Handling Rules, which make a linear-logic declarative semantics of CHR a promising approach. Furthermore, intuitionistic logic can be embedded into (intuitionistic) linear logic, which will be an indispensable feature in our semantics. Our approach is somewhat similar to the ones taken in [7] and [5] in that we will define a linear-logic semantics for an existing programming paradigm.

This paper is structured as follows: Section 2 will give a short introduction to the segment of *intuitionistic linear logic*. In Sect. 3, *constraint handling rules* will be presented with a particular focus on declarative semantics. It will become clear what the limitations to the classical declarative semantics are, which we hope to overcome by using linear logic. Section 4 will introduce our linear-logic semantics for CHR, explain its benefits and present strong theorems concerning soundness and completeness of the linear-logic declarative semantics w.r.t. operational semantics. In Sect. 5 we will give an example for the application of our proposed semantics. A conclusion will be given in Sect. 6.

2 Intuitionistic Linear Logic

Intuitionistic linear logic (**ILL**) is a subset of linear logic [6, 10, 12, 14] which is constituted by the symbols $\otimes$, &, $\oplus$, $\multimap$ and ! as well as the constants 1, $\top$ and 0. In the following a short explanation of its symbols will be given.

2.1 Connectives

Let us take a look at an easy example first:

$$A \multimap B$$

The above formula is an example for linear implication. It is pronounced *"consuming A yields B"*. Since the idea of "consuming" logical truth (in the classical sense) somewhat stresses the imagination, linear logic is generally considered as dealing with resources rather than with propositions.

The meaning represented by the symbol $\otimes$ (*"times"*) is reasonably close to the intuitive grasp we usually have of the classical conjunction $\wedge$. Which is, that both formulas connected hold at the same time. Consequently, the expression $A \otimes B$ is pronounced *"both A and B"*.

Note that an implication of the form $A \multimap B$ allows us to consume-produce exactly once, in the process of which the implication *itself* is used up. E.g. in classical first-order logic the following holds:

$$A \wedge (A \rightarrow B) \vdash B \wedge (A \rightarrow B)$$

On the contrary, the following is *not* true:

$$A \otimes (A \multimap B) \vdash B \otimes (A \multimap B)$$

The following conclusion is correct:

$$A \otimes (A \multimap B) \vdash B$$

The connective & (*"with"*) represents an option of (internal) choice. The formula $A\&B$ is pronounced *"choose from either A or B* and allows us to infer either A or B (but not both, which would be $A \otimes B$).

Similar to the *with* conjunction, the connective $\oplus$ *"plus"* also denotes an alternative. However, the choice is external, i.e. if the formula $A \oplus B$ holds, then *either* A or B will hold (but not both!), although it is not stated which one. The formula $A \oplus B$ is consequently pronounced *"either A or B"*.

We mentioned before that linear logic is considered as discussing resources rather than stable facts. Nevertheless, it is extremely useful if not indispensable to have an option for stable truth (in the classical sense) to interact with variable truth (i.e. resources). This is provided for by the ! (*"bang"*) symbol.

In linear logic, the *bang* marks either a stable truth or an abounding resource that – within the boundaries of the model – cannot get used up (which essentially boils down to the same thing). A typical application for the bang is in implications that can be applied an unlimited number of times. It is thus correct to conclude the following:

$$A \otimes !(A \multimap B) \vdash B \otimes !(A \multimap B)$$

There are three ways to *actualize* a banged resource's potential, namely *weakening*, *contraction* and *dereliction* [10].

Dereliction designates the possibility to use a banged resource just like an un-banged instance.

$$\vdash !A \multimap A \ (dereliction)$$

Contraction denotes the fact, that we may duplicate any banged resource, time and again, to potential infinity.

$$\vdash !A \multimap !A \otimes !A \ (contraction)$$

Weakening is the property of a banged resource that – just like a classical proposition – it needs not be used at all and may just be omitted.

$$\vdash !A \multimap 1 \ (weakening)$$

Furthermore, the following equivalence holds:

$$!(A \& B) \dashv\vdash (!A \otimes !B)$$

2.2 Constants and Quantification

We will furthermore consider two constants: 0 (*zero*) and $\top$ (*top*). The constant $\top$ represents the goal in favor of which every resource can be consumed. As for an intuition, we may think of it as a trash can.

As for the 0: In classical logic, there is the principle *"ex falso, quod libet*, i.e. from a proposition that equals *false*, we can deduce any other proposition. This aspect of falsity is represented by 0, which by definition yields every resource. In this sense, 0 represents *impossibility*.

Just like classical logic, linear predicate logic offers the quantifiers $\forall$ and $\exists$. Since we cannot directly convey the classical concept of *truth* to linear logic, we will use the term *provability* instead. The proposition $\exists x Q(x)$ is provable if there is a term t for which $[t/x]Q(x)$ is provable. The proposition $\forall x Q(x)$ is provable, if $[a/x]Q(x)$ is provable for a new parameter a about which we can make no assumption. For convenience, we can define x to range over a domain $\mathbb{D}$.

The following equations hold:

$$\forall x Q(x) \equiv \mathop{\&}_{x \in \mathbb{D}} Q(x)$$

$$\exists x Q(x) \equiv \bigoplus_{x \in \mathbb{D}} Q(x)$$

2.3 Girard Translation

Among the key features of intuitionistic linear logic is the possibility to faithfully translate (classical) intuitionistic logic into intuitionistic linear logic while preserving the full power of the former. Fig. 1 presents one of several possible translations, called *Girard Translation* [10], in the notation of [12].

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

Fig. 1. Translation $^+$ from intuitionistic logic into linear logic

3 CHR

CHR is a concurrent committed-choice constraint programming language, developed in the 1990s for the implementation of constraint solvers. It is traditionally used as an extension to other programming languages – especially constraint logic programming languages – but has been used increasingly as a general-purpose programming language in the recent past. In this section we will give an overview of its syntax and operational semantics as well as its classical declarative semantics [1, 8, 2].

3.1 CHR Syntax

Constraints are predicates of first-order logic. In CHR, there are two notably different types of constraints, which we will refer to as *built-in constraints* and *CHR constraints*. CHR constraints, will be handled by a CHR program whereas built-in constraint are predefined in the CHR implementation.

Definition 1. *An* atomic built-in constraint *is an expression of the form* $c(t_1, ..., t_n)$*, where* c *is an* n*-ary constraint symbol and* $t_1, ..., t_n$ *are terms. A* built-in constraint *is either an atomic built-in constraint or a conjunction of built-in constraints.*

A CHR constraint *is a non-empty multiset, the elements of which have the form* $e(t_1, ..., t_n)$*, where* e *is an* n*-ary constraint symbol and* $t_1, ..., t_n$ *are terms. A CHR constraint is called* atomic *if it has exactly one element.*

Note that the syntactic equality constraint $=$ as well as the propositions *true* and *false* are built-in by definition.

Definition 2. *A* goal *is either* $\{\top\}$ *(top),* $\{\bot\}$ *(bottom), an expression of the form* $\{C\}$ *– where* C *is an atomic built-in constraint –, a CHR constraint or a multi-set union of goals.*

Apart from definitions, we leave away the curly brackets from both CHR constraints and goals.

A CHR program consists of a set of rules, determining the transformation of constraints. These rules are the *constraint handling rules*, i.e. the CHR, of which we distinguish two types: **Simplify** and **Propagate** . A **Simplify** rule determines the replacement of a CHR constraint, usually a subset of a larger goal, with a multiset of simpler constraints whereas a **Propagate** rule augments an existing goal by one or several elements (which hopefully leads to further simplification later on).

Definition 3. *A* simplification rule *is of the form* $H \Leftrightarrow G|B$*. A* propagation rule *is of the form* $H \Rightarrow G|B$*, where the* head H *is a CHR constraint, the* guard G *is a built-in constraint and the* body B *is a goal.*

A CHR program *is a finite set of rules.*

3.2 CHR Operational Semantics

Note that the operational semantics defined here is not necessarily identical to the behavior of an actual implementation.

Definition 4. *A state is a pair* $\langle G; C \rangle$*, where* G *is a goal and* C *is a built-in constraint.*

Of the two components, only the goal store G is directly accessible by CHR, i.e. only elements stored here will be transformed by constraint handling rules. The built-in constraint store C is not directly accessible, i.e. CHR can add (built-in) constraints to the store, but cannot manipulate or remove its elements.

Definition 5. *The* constraint theory *CT is a non-empty, consistent first-order theory over the built-in constraints, including syntactic equality =, as well as the propostions* true *and* false*.*

The constraint theory CT is implicitly realized by the predefined constraint handlers.

At runtime, a CHR program is provided with an initial state and will be executed until either no more rules are applicable or a contradiction occurs in the constraint store (which will result in the constraint store equaling *false*).

Definition 6. *An* initial state *is of the form* $\langle G; true \rangle$*. A* failed final state *is of the form* $\langle G; false \rangle$*. A state is called a* successful final state *if it is of the form* $\langle E; C \rangle$ *with no transition applicable.*

Initial states are distinguished from states that appear in a derivation, since declarative semantics will assign a different logical reading to either type of state.

Definition 7. *A* derived state *is a state* S_a *which appears in a derivation from an initial state* S_0*. The variables* $\bar{x}_a$ *that appear in* S_a *but not in* S_0 *are called* local variables *of* S_a*.*

The transition rules in Fig. 2 describe the transition relation. Note that we omit the **Solve** transition here since it is irrelevant to our cause.

Simplify	
If	$(F \Leftrightarrow D \vert H)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall(C \rightarrow \exists \bar{x}(F = E \wedge D))$
then	$\langle E \cup G; C \rangle \mapsto \langle H \cup G; (F = E) \wedge D \wedge C \rangle$
Propagate	
If	$(F \Rightarrow D \vert H)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall(C \rightarrow \exists \bar{x}(F = E \wedge D))$
then	$\langle E \cup G; C \rangle \mapsto \langle E \cup H \cup G; (F = E) \wedge D \wedge C \rangle$

Fig. 2. CHR transition rules

The sequence $\bar{x}$ represents the variables in $(F \Leftrightarrow D|H)$. We require always a fresh variant of a rule $(F \Leftrightarrow D|H)$, i.e. that all variables are given unique new names. The CHR rule's head F must be matched (pairwise) with CHR constraints E from the goal store. The constraints in C and D as well as = are built-in constraints and thus are handled according to the constraint theory CT. On application of the rule, the constraint store is augmented by the matching $(F = E)$ as well as the guard D.

3.3 The (Classical) Declarative Semantics of CHR

Figure 3 defines the first-order-logic declarative semantics of CHR. In the transformations of CHR rules, $\bar{y}$ represents the variables that *only* appear in the body G of the rule. While these variables are existentially quantified, all other variables become universally quantified.

Built-in constraints:	C'	::=	C
CHR constraints:	$\{e(t_1,...,t_n)\}'$	::=	$e(t_1,...,t_n)$
	$(E \cup F)'$	::=	$E' \wedge F'$
Goals:	$\{\top\}'$	::=	$\top$
	$\{\bot\}'$	::=	$\bot$
	$\{c(t_1,...,t_n)\}'$	::=	$c(t_1,...,t_n)$
	$(G \cup H)'$	::=	$G' \wedge H'$
Initial states:	$\langle G; true\rangle'$	::=	G'
Derived states:	$\langle G; C\rangle'$	::=	$\exists \bar{x}_a(G' \wedge C')$
Simplify rules:	$(E \Leftrightarrow C \mid G)'$	::=	$\forall(C \rightarrow (E \leftrightarrow \exists \bar{y} G))$
Propagate rules:	$(E \Rightarrow C \mid G)'$	::=	$\forall(C \rightarrow (E \rightarrow \exists \bar{y} G))$
Programs:	$(R_1 ... R_m)'$	::=	$R'_1 \wedge ... \wedge R'_m$

Fig. 3. Classical-logic declarative semantics P' of a program P

3.4 Soundness and Completeness

The first-order-logic semantics given in Fig. 3 maps every CHR program P to a set of logical formulae P' which form a mathematical theory. The following theorems will show that the operational and this declarative semantics are strongly related.

Definition 8. *A* computable constraint *of a state S_0 is the logical reading S'_a of a derived state of S_0. An* answer (constraint) *of a state S_0 is the logical reading S'_n of a final state of a derivation from S_0.*

The following theorems are proved in [2]:

Theorem 1. (Soundness). *Let P be a CHR program and S_0 be an initial state. If S_0 has a derivation with answer constraint S'_n, then $P' \cup CT \models \forall(S'_0 \leftrightarrow S'_n)$.*

Theorem 2. (Completeness). *Let P be a CHR program and S_0 be an initial state with at least one finite derivation. If $P' \cup CT \models \forall(S'_0 \leftrightarrow S'_n)$, then S_0 has a derivation with answer constraint S'_ν such that $P' \cup CT \models \forall(S'_\nu \leftrightarrow S'_n)$.*

4 A Linear-Logic Semantics for CHR

CHR is a powerful and flexible tool for writing not only constraint handlers but also general-purpose concurrent programs. As far as constraint handlers are concerned, there is a useful and consistent declarative semantics. However, when used as a general-purpose programming language and program rules go beyond a mere representation of a mathematical theory, programs tend to produce inconsistent logical readings as has been examined e.g. in [2].

In this section we will discuss the limitations of the classical declarative semantics. Then we will propose a declarative semantics for CHR which is based on intuitionistic linear logic and we will show it can provide a consistent logical reading for non-traditional CHR programs. We will also state two theorems proving the soundness and completeness of our approach.

4.1 Limitations of the Classical Declarative Semantics

In Sect. 1 we already gave an example for a CHR program with an inconsistent logical reading with respect to the classical declarative semantics. Below another such program is given to further illustrate the matter.

Example 2. The program given below applies the Sieve of Eratosthenes to an interval of cardinal numbers in order to "sieve out the prime numbers from that interval.

```
candidate(N) ⇔ N>1 | M is N-1, prime(N), candidate(M)   (r1)
candidate(1) ⇔ true                                      (r2)
prime(M), prime(N) ⇔ M mod N =:= 0 | prime(N)            (r3)
```

The program implements two constraints: `candidate` and `prime`. The `candidate` constraint is to create the set of numbers on which to work, represented as individual constraints. The actual sieving is perfomed by the `prime` constraint. The program is executed with the goal `candidate(N)` in the initial state, where N is the upper limit of the interval on which to work.

Consider the declarative semantics of the constraint *prime*:

$$\forall(M \bmod N = 0 \rightarrow (prime(M) \wedge prime(N) \leftrightarrow prime(N))$$

What this logical expression actually says is that "a number is prime, if it is a multiple of another prime number (sic!). The problem is that the `prime` constraint does not consist of only static information. Its input is an initial range of cardinal numbers representing candidates for primes. Only upon completion of the calculation they do represent the actual primes. Predicate logic has no straightforward means to express this dynamics.

4.2 An Intuitionistic Linear-Logic Semantics

The obvious similarity between linear implication and CHR constraint substitution as well as the possible representation of multiplicities and embedding of intuitionistic logic make linear logic a likely candidate for providing a suitable declarative semantics.

In this section we introduce an intuitionistic linear logic (cf. Sect. 2.3) semantics of CHR. Figure 4 shows the proposed semantics. It adheres to some extent to the classical declarative semantics. The main differences are the interpretation of CHR constraints as linear resources (and that of built-in constraints as embedded intuitionistic propositions), as well as the distinctly different logical reading of CHR rules as expressing linear implementation rather than logical equivalence.

Built-in constraints:	$c(t_1,...,t_n)^L$	::=	$!c(t_1,...,t_n)$
	$(C \wedge D)^L$	::=	$C \otimes D$
CHR constraints:	$\{e(t_1,...,t_n)\}^L$	::=	$e(t_1,...,t_n)$
	$(E \cup F)^L$	::=	$E^L \otimes F^L$
Goals:	$\{\top\}^L$	::=	$\top$
	$\{\bot\}^L$	::=	0
	$\{c(t_1,...,t_n)\}^L$	::=	$c(t_1,...,t_n)$
	$(G \cup H)^L$	::=	$G^L \otimes H^L$
Initial states:	$S_0^L = \langle G; true\rangle^L$	::=	G^L
Derived states:	$S_a^L = \langle G; C\rangle^L$	::=	$\exists \bar{x}_a (G^L \otimes C^L)$
Simplify rules:	$(E \Leftrightarrow C \mid G)^L$	::=	$!\forall \left((!C^L) \multimap \left(E^L \multimap \exists \bar{y} G^L\right)\right)$
Propagate rules:	$(E \Rightarrow C \mid G)^L$	::=	$!\forall \left((!C^L) \multimap \left(E^L \multimap E^L \otimes \exists \bar{y} G^L\right)\right)$
Programs:	$(R_1 ... R_m)^L$	::=	$R_1^L \otimes ... \otimes R_m^L$

Fig. 4. Linear-logic declarative semantics P^L of a program P

We assume that built-in constraints are propositions of intuitionistic logic, translated according to Girard Translation as introduced in Sect. 2.3. States are handled much the same as in classical declarative semantics: The logical reading of an initial state is again the logical reading of the goal. The logical reading of a derived state S_a is again a conjunction, now a $\otimes$ conjunction, of its components' readings with its local variables existentially quantified.

A **Simplify** rule $(E \Leftrightarrow C \mid G)$ maps to $!\forall \left((!C^L) \multimap \left(E^L \multimap E^L \otimes \exists \bar{y} G^L\right)\right)$, where $\bar{y}$ represents the variables that *only* appear in the body G of the rule. As before, the fulfillment of the guard is a premise. Instead of equivalence between head and body, however, it implies now that consuming the head produces the body. Note that the formula is banged, since it is to be used not only once, of course. A **Propagate** rule follows the same pattern. The only difference is that here, consuming the head produces the head *and* the body.

Example 3. We will take another look at Example 2 and see how its declarative semantics benefits from the linear-logic approach. This is what the **ILL** reading looks like (for the constraint "prime).

$$!\forall\,((M \;mod\; N =:= 0) \multimap (prime(M) \otimes prime(N) \multimap prime(N)))$$

As we can see, this reading is no longer inconsistent with the mathematical understanding of prime numbers. It is indeed rather a suitable **ILL** representation of the program's workings.

Example 4. The improvement regarding the coin-throw example mentioned in Sect. 1 is quite alike. The **ILL** reading for that program is:

$$throw(Coin) \multimap Coin = head$$
$$throw(Coin) \multimap Coin = tail$$

This is logically equivalent to the following:

$$!\,(throw(Coin) \multimap (Coin = head)\&(Coin = tail))$$

The above reads as: *Of course, consuming* Throw(Coin) *produces: Choose from* (Coin = head) *and* (Coin = tail). Thus, our logical reading implies internal, committed choice.

4.3 Soundness and Completeness

Concerning soundness, our approach is analogous to that one used in the classical framework, (cf. Sect. 3.4) relying basically on Lemma 1 which proves that all computable constraints of a state S_0 are linearly implied by the initial state's logical reading.

The constraint theory CT, which we require to be of intuitionistic logic, is translated according to the *Girard Translation* (cf. Sect. 2.3).

Lemma 1. *Let P be a program, P^L its linear-logic reading, S_0 be a state. If S_n is a computable constraint of S_0 then:*

$$P^L, !CT^+ \models \forall(S_0^L \multimap S_n^L)$$

From this lemma, Theorem 3 follows directly.

Theorem 3. (Soundness). *Let P be a CHR program and S_0 be an initial state. If S_0 has a derivation with answer S_n^L, then $P^L, !CT^+ \models \forall(S_0^L \multimap S_n^L)$.*

We also have a surprisingly strong completeness theorem.

Theorem 4. (Completeness). *If S_0 and S_n are states, such that $P^L, !CT^+ \vdash S_0^L \multimap S_n^L$ then S_0 has a derivable constraint S_ν such that $!CT^+ \vdash S_\nu^L \multimap S_n^L$.*

The complete proofs for both theorems can be found in [4]. Whereas the proofs for Lemma 1 and Theorem 3 parallel the respective proofs in the classical-logic case, the proof for Theorem 4 follows a unique approach, which is sketched below.

Proof Sketch. The proof of Theorem 4 consists of three parts [4].

The **first part** establishes a series of lemmas in order to transform the expression $P^L, !CT^+ \vdash S_0^L \multimap S_n^L$ into an equivalent form that is easier to work with. This transformation involves bringing the formula S_0^L to the precondition side, stripping the expression of bang symbols and finally removing quantifiers. The most difficult task is the removal of the bang symbols. We consider a cut-free proof of the original expression. We can show by structural induction that for at least one bang-free version of that expression a (cut-free) proof must exist.

At the end of the first part, we have transformed our expression $P^L, !CT^+ \vdash S_0^L \multimap S_n^L$ into the equivalent form $\overline{P^L}, \overline{!CT^+}, S_0^L \vdash S_n^L$, where the horizontal bar marks the removal of all bangs and quantifiers [2] . Since there are no more bangs, all rules in $\overline{P^L}$ and $\overline{!CT^+}$ appear in certain multiplicities, according to how often each rule is applied.

In the **second part** we force the transformed expression to act similar to a CHR program, i.e. we prove by structural induction that there must be at least one implication in either $\overline{P^L}$ or $\overline{!CT^+}$ of the form $(A \multimap B)$ where A is a conjunction of atoms that is contained in S_0^L, so the implication can be applied to S_0^L. Assuming that $(A \multimap B)$ is in $\overline{P^L}$, this models the application of a CHR rule on the constraint store. Otherwise it corresponds to a rule of the constraint theory CT. By repeated application of the above reasoning we can force the application of all implications in $\overline{P^L}$.

Having shown this, we are already quite close to our goal. We can now safely say that the logical transition from $\overline{P^L}, \overline{!CT^+}, S_0^L$ to $\overline{S_n^L}$ can be cut into smaller steps, *similar* to the steps of a CHR program. Actually, the only difference is in the built-in constraints: a CHR computation neither allows the consumption of a built-in constraint nor the inference of a built-in constraint that is *unnecessary* in that it does not lead to another CHR rule to become applicable.

This final problem is dealt with in the **third part** where we prove that our logically derived expression S_n^L is so close to an actually derivable constraint S_ν^L that the former can be inferred from the latter by applying the constraint theory CT only, i.e. $!CT^+ \vdash S_\nu^L \multimap S_n^L$. This is done by a methodically simple, yet formally tedious induction over the transition steps identified in the second part of the proof.

[2] As $!(A\&B) \dashv\vdash (!A \otimes !B)$, the expression $\overline{!CT^+}$ is *not* equivalent to CT^+.

5 Example: Union-find in CHR

As CHR is increasingly being used as a general-purpose concurrent constraint programming language, focus has shifted to the question whether it can be used to implement classic algorithms in an efficient and elegant way. This has successfully been done for Tarjan's union-find algorithm in [9]. However, in that paper it has also been shown that this algorithm has a destructive update which cannot adequately be modeled in classical logic. We will show here how the linear-logic declarative semantics can provide a solution.

5.1 The Union-Find Algorithm in CHR

The original union-find algorithm was introduced by Tarjan in [13]. It serves to maintain collections of disjoint sets where each set is represented by an unambiguous representative element. The structure has to support the three operations:

- `make(X)`: create a new set with the single element `X`.
- `find(X)`: return the representative of the set in which `X` is contained
- `union(X,Y)`: join the two sets that contain `X` and `Y`, respectively (possibly destroying the old sets and changing the representative).

In the basic algorithm discussed here the sets are represented by rooted trees, where the roots are the respective representative elements. Trees are represented by the constraints `A ~> B` and `root(A)`. The three operations are implemented as follows.

- `make(X)`: generate a new tree with the only node `X`.
- `find(X)`: follow the path from node `X` to the root by repeatedly going to the parent. Return the root as representative.
- `union(X,Y)`: find the representatives of `X` and `Y` and link them by making one root point to the other root.

The following CHR program implements the Union-Find Algorithm [9].

```
make(A) ⇔ root(A)                                  (make)
union(A,B) ⇔ find(A,X), find(B,Y), link(X,Y)       (union)

A ~> B, find(A,X) ⇔ A ~> B, find(B,X)              (findNode)
root(A), find(A,X) ⇔ root(A), X=A                  (findRoot)

link(A,A) ⇔ true                                   (linkEq)
link(A,B), root(A), root(B) ⇔ B ~> A, root(A)      (link)
```

5.2 Declarative Semantics

Concerning logical correctness we will limit ourselves to the `link` rule because it is here where the problem arises. The classical declarative reading for this rule reads as follows:

$$link(A, B) \wedge root(A) \wedge root(B) \Leftrightarrow B \rightsquigarrow A \wedge root(A)$$

The reading as given above establishes a supposed logical equivalence where the node B is a root and a non-root at the same time (*root(B)* and $B \rightsquigarrow A$ hold), but actually a destructive update from a root to a non-root takes place. The problem is in principle the same as was presented in Sect. 4: Classical logic is able to deal with static truth only and has no capabilities to represent dynamic processes without resorting to explicit representation of time. In contrast, the linear-logic reading of the respective constraint reads as follows:

$$!\,(link(A, B) \otimes root(A) \otimes root(B) \multimap (B \rightsquigarrow A) \otimes root(A))$$

The above can be read as: *Of course, consuming all of* link(A,B)*,* root(A) *and* root(B) *yields both* B $\rightsquigarrow$ A *and* root(A). Or less formally: On the condition that both *root(A)* and *root(B)* hold, *link(A,B)* triggers the change of *root(A)* to $B \rightsquigarrow A$. This reading directly expresses the dynamic update process which is taking place.

This example shows how our linear-logic semantics can provide logical readings for non-traditional CHR programs in cases where there is no consistent reading with respect to the classical semantics. Thus, the process of proving logical correctness for CHR programs is considerably simplified.

6 Conclusion

We have developed a linear-logic semantics for CHR as an alternative to the classical declarative semantics. The new declarative semantics is based on the segment of intuitive linear logic.

We have shown that this declarative semantics indeed overcomes the limitations of the classical declarative semantics, which originally motivated this work. The new semantics features surprisingly strong theorems on both soundness and completeness, thus simplifying the process of proving logical correctness of CHR programs. Details can be found in [4]. How well this can be done in practice, and what insights it offers, remains a topic for future work.

Since this is the first paper relating CHR to linear logic, there are numerous options for further work in this field. An obvious follow-up project would be a thorough comparison of CHR to related works such as the LCC class of linear concurrent constraint programming languages [7], for which a linear-logic semantics exists as well.

In the program presented in Sect. 5, a large part of the program actually does have a consistent classical reading. In a case like this it might be more convenient

to apply our linear-logic semantics only on those parts of the program where the classical semantics produces inconsistent results, in order to get to a more intuitive logical reading. To this end, it is necessary to more closely inspect the relationship between classical and linear-logic readings. Classical program parts could be identified by a modified confluence analysis, since confluence implies consistency of the classical-logic reading of a program [2].

Our linear-logic semantics for CHR may also shed light on executable subsets of linear logic and the related recent separation logic. An interesting approach would be to develop a CHR constraint handler for a larger segment of linear logic than that which is actually used in the declarative semantics. This would be an approach closer to the ones taken in [11] and [3].

References

1. Slim Abdennadher, Thom Frühwirth: *Essentials of constraint programming*. Springer, 2003.
2. Slim Abdennadher, Thom Frühwirth, Holger Meuss: *Confluence and semantics of constraint simplification rules*. Constraints 4(2):133-165 (1999).
3. Jean-Marc Andreoli, Remo Pareschi: *LO and Behold! Concurrent Structured Processes*. ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, 25(10):44-56, October 1990.
4. Hariolf 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`
5. Marco Bozzano, Giorgio Delzanno, Maurizio Martelli: *A Linear Logic Specification of Chimera*. DYNAMICS '97, a satellite workshop of ILPS'97, Proceedings, 1997.
6. Vincent Danos, Roberto Di Cosmo: *Initiation to Linear Logic*. Course notes, June 1992.
7. François Fages, Paul Ruet, Sylvain Soliman: *Linear Concurrent Constraint Programming: Operational and Phase Semantics*. Information and Computation, 165(1):14-41, 2001.
8. Thom Frühwirth: *Theory and practice of constraint handling rules*. Journal of Logic Programming, 37(1-3):95-138, 1998.
9. Tom Schrijvers, Thom Frühwirth: *Optimal Union-Find in Constraint Handling Rules - Programming Pearl*. Theory and Practice of Logic Programming (TPLP), to appear 2005.
10. Jean-Yves Girard: *Linear Logic: Its syntax and semantics*. Theoretical Computer Science, 50:1-102, 1987.
11. James Harland, David Pym, Michael Winikoff: *Programming in Lygon: an overview*. Algebraic Methodology and Software Technology (AMAST 96), 5th International Conference, Proceedings, 391-405, 1996.
12. Frank Pfenning: *Linear Logic*. Material for the homonymous course at Carnegie Mellon University. Draft of 2002.
13. Robert E. Tarjan, Jan van Leeuwen. *Worst-case analysis of set union algorithms*. Journal of the ACM, 31(2):245-281, 1984.
14. Philip Wadler: *A taste of linear logic*. Invited talk, Mathematical Foundations of Computing Science, Springer LNCS 711, 1993.