Operational Semantics of Constraint Logic Programs with Coroutining [1]

Andreas Podelski[2]
Max-Planck-Institut für Informatik
Im Stadtwald, D-6123 Saarbrücken, Germany
podelski@mpi-sb.mpg.de

Gert Smolka
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
smolka@dfki.uni-sb.de

Abstract

The semantics of constraint logic programming languages with coroutining facilities ("freeze," suspension, residuation, *etc.*) cannot be fully declarative; thus, an operational semantics has to be taken as the defining one. We give a formal operational semantics for a Prolog-like language with cut and entailment-based conditional. The difficulty here is to present the semantics in a form that abstracts away inessential details and highlights the interaction between language constructs. Our approach is derived from those used for concurrent calculi. We use abstract syntax trees, congruence laws and rewrite rules to define the semantics. A computation step is modeled as the application of a rewrite rule to an abstract syntax tree modulo structural congruence. This semantics serves as a defining tool for the language designer and as the interface between the language designer and implementor; it allows the programmer to check his intuition with a formal execution model and it gives him a performance measure for the execution of programs. We have used the semantics to make precise, for the first time, the critical interaction between sequential execution (including backtracking and cut pruning) and coroutining. In particular we exhibit cases where this interaction can lead to indeterministic results (*i.e.*, to non-predictable program execution).

[1] Appeared in Proceedings of International Conference of Logic Programming (ICLP'95), Kanagawa, Japan, June 1995. Pages 449–463, editor Leon Sterling. MIT Press.

[2] partially supported by ESPRIT working group Nr. 8666 CONTESSA

1 Introduction

The logical semantics (" $\models \equiv \vdash$") which has been given for logic programming languages [7, 6] has been one reason for their success. It presents these languages as executable specification languages. The declarative semantics becomes practically useful when, for example, programming search problems. These languages are, however, mainly used as programming languages which have declarative as well as purely operational functionalities (such as the ordering of goals, the cut operator, and facilities for coroutining which appear in one form or another[3] in most CLP- or Prolog-Systems. Also, it is well known that the formal correspondence between declarative and operational semantics is lost even for the "pure" subset of practical language implementations. Thus, the semantics cannot be fully declarative, and we have to take an operational semantics as the defining one.

To achieve the definition of an operational semantics is not difficult in principle; one need only formalize an interpreter. The difficulty is to present the semantics in a form that abstract away inessential details and highlights the interaction between language constructs. The work in [3] gives the operational semantics of "freeze," but on a very low-level; for example, it formalizes the stepwise search through the list of frozen goals. Such a low-level approach seems suitable for specifying and comparing language implementations, but not so much for designing and specifying the languages themselves. On a very high-level, declarative and operational semantics for comitted-choice languages have been given by Maher in [8], who captured guard satisfaction as constraint entailment. Similar in spirit is the work in [14] which considers a logic programming language with don't-know choice and guarded rules.

In this paper, we give a short, simple and formally precise description of the operational semantics of logic programs with coroutining facilities. In comparison with [3], our semantics is high-level. In contrast to [3], it treats constraint handling logically . In contrast to [8, 14], our semantics accounts for order (of goals and alternatives), cut and eager wakening and ignores declarative semantics. Our approach is based on *abstract-tree rewriting* which appeared in a revised representation of the π-calculus [9] and was employed in [15, 16] to formulate calculi for higher-order concurrent constraint programming model. The abstract-tree rewriting model proves particularly useful for constraint programming since constraint propagation and simplification (for testing satisfiability as well as entailment) can be accommodated elegantly by means of the congruence on trees. This congruence is defined using the logical meaning of constraints. Thus, we model constraints logically, as is standard in CLP.

The next section introduces our general approach. Section 3 introduces the notions of constraints and predicate definitions and applications and considers simple programs, which are built up with the operators " $\&$ " and " ; ". Sections 4, 5 and 6 consider programs obtained by adding an operator, either

[3]For example: "freeze," suspension, residuation (negated constraints also introduce an implicit form of coroutining); *cf.* [5, 10, 11, 4, 1]).

"!", " ?" or "$\exists$", to simple programs. Section 7 considers the combination of all five operators. The article ends with a conclusion section.

2 The Approach

The ingredients of our semantics are (*cf.*, Figure 1): BNF style rules to define the abstract syntax, reduction rules (" $\longrightarrow$ ") for the operational and congruence laws (" $\equiv$ ") for the declarative aspects.

The execution of a program is seen as a sequence of state transitions. The states are modeled by formulae ("continuations") which are represented as trees as usual. A computation step is modeled as the application of a rewrite rule to an abstract syntax tree modulo structural congruence. Formally, the reduction relation " $\Longrightarrow$ " modeling the direct state transitions is an extension of the relation $\longrightarrow$ given by the reduction rules, namely

$$\frac{C_1 \ \equiv \ C_1', \ \ C_2 \ \equiv \ C_2', \ \ C_1' \ \longrightarrow \ C_2'}{\mathcal{C}[C_1] \ \Longrightarrow \ \mathcal{C}[C_2]}$$

for any *context* $\mathcal{C}$ (of the form defined in the corresponding definition) and configurations C_1, C_1', C_2, C_2'. A context is a "tree with a hole;" $\mathcal{C}[C]$ denotes the tree obtained from $\mathcal{C}$ by putting the tree C into the hole. For example, for the contexts defined in Figure 1 we define $\bullet[C] = C$ and $(\bullet\, ;\, C')[C] = C\, ;\, C'$.

The trees are made abstract by a congruence. The idea is that congruent trees describe the same state. Formally, the relation $\equiv$ is the congruence generated by the congruence laws.[4]

The purpose of congruences is twofold: (1) They serve to make trees sufficiently abstract to suitably model states. For example, $E_1 \wedge (E_2 \wedge E_3)$ and $(E_1 \wedge E_2) \wedge E_3$ are different trees; by making them congruent one can talk about the list of conjuncts written as $E_1 \wedge E_2 \wedge E_3$. (2) They serve to describe the effect of a functionality of the language on a high level (*i.e.*, declaratively): Instead of modeling the effect through transitions between certain states, one identifies these states through the congruence. For example, we describe the effect of unification by a congruence law (Simplification). The implementation of the language has to ensure that these "hidden" transitions are possible when needed.

3 Simple Programs ("&" and " ; ")

We will motivate and explain the definition for the execution of simple programs in Figure 1 at hand of an example. We take the definition of the length predicate, `length([],zero). length([HT],succ(N))` :- length(T,N). — and, hereby indicating how the systematic translation from concrete into

[4]This means, $\equiv$ is the least equivalence relation which respects the congruence laws and the rule: If $t_1 \ \equiv \ t_1'$ and $t_2 \ \equiv \ t_2'$, then $t_1 \circ t_2 \ \equiv \ t_1' \circ t_2'$ for binary tree-composing symbols $\circ$, and accordingly for ternary ones.

SYNTAX

Expressions	$E \ ::= \ \varphi \ \mid \ p\bar{x} \ \mid \ E_1 \,\&\, E_2 \ \mid \ E_1\,;\,E_2$
Configurations	$C \ ::= \ \varphi\!:\!E \ \mid \ \bot \ \mid \ C_1\,;\,C_2$
Contexts	$\mathcal{C} \ ::= \ \bullet \ \mid \ \mathcal{C}\,;\,C$

REDUCTION RULES

Application Elaboration	$\varphi\!:\!p\bar{x} \,\&\, E \ \longrightarrow \ \varphi\!:\!\bar{x}=\bar{y} \,\&\, E_1 \,\&\, E$ if $\mathrm{p\bar{y}} \leftrightarrow \mathrm{E_1}$ in program, $\mathcal{V}(\varphi\!:\!\mathrm{E}) \cap \mathcal{V}(\mathrm{p\bar{y}} \leftrightarrow \mathrm{E_1}) = \emptyset$
Constraint Elaboration	$\varphi\!:\!\varphi_1 \,\&\, E \ \longrightarrow \ \varphi \wedge \varphi_1 \!:\! E$ if $\Delta \not\models \varphi \wedge \varphi_1 \leftrightarrow \bot,\ \varphi_1 \neq \top$ $\longrightarrow \ \bot$ if $\Delta \models \varphi \wedge \varphi_1 \leftrightarrow \bot$
Or Elaboration	$\varphi\!:\!(E_1\,;\,E_2) \,\&\, E \ \longrightarrow \ (\varphi\!:\!E_1 \,\&\, E)\ ;\ (\varphi\!:\!E_2 \,\&\, E)$

CONGRUENCE LAWS

Sequential And	$(\{\mathit{Expressions}\},\ \&\,,\ \top)$ is a monoid.
Sequential Or	$(\{\mathit{Configurations}\},\ ;\,,\ \bot)$ is a monoid.
Simplification	$\varphi_1 \ \equiv \ \varphi_2$ if $\Delta \models \varphi_1 \leftrightarrow \varphi_2$.

Figure 1: Simple Programs ("&" and ";")

abstract syntax works, translate it to the following syntactic formula.[5]

$$\begin{aligned} \mathit{length}(x,n) \leftrightarrow (\ & (x = \mathit{nil} \wedge n = \mathit{zero})\,; \\ & ((x = \mathit{cons}(y,z) \wedge n = \mathit{succ}(m)) \,\&\, \mathit{length}(z,m))\ ) \end{aligned}$$

We say that the predicate $\mathit{length}(x,n)$ is defined by an *expression* E which is composed of sub-expression with the symbols ";" (read: "Sequential Or") and "&" (read: "Sequential And"). The atomic sub-expressions are either *applications* $p\bar{x}$ (here: $\mathit{length}(z,m)$) or *constraints* φ (here: $x = \mathit{cons}(y,z) \wedge n = \mathit{succ}(m)$). Constraints are themselves composed of atomic constraints with the symbol "$\wedge$" (read: "Logical And").[6]

In order to motivate the use of two "And" symbols now, we note that the difference between the two expressions $x = f(y,a) \wedge y = a$ and $x = f(y,a) \,\&\, y = a$ corresponds to the difference between `X=f(a,a)` and `X=f(Y,a), Y=a` in Prolog syntax.[7] In connection with coroutines, this difference has an operational significance (if the constraint $y = a$ fails, coroutines may be fired in the second case, but not in the first; *cf.*, Section 5).

The semantics is formally parametrized with a *constraint system*, which defines the syntax and the interpretation of the particular class of first-order

[5] This is not yet a logical definition; *i.e.*, the symbol "$\leftrightarrow$" does, for now, not have the logical reading "if and only if." We omit the account of existential quantification until Section 6.

[6] We use $\bar{x}$ as an abbreviation for a tuple of pairwise different variables.

[7] The formally correct expression corresponding to `X=f(a,a)` is $\exists y \exists z(x = f(y,z) \wedge y = a \wedge z = a)$.

logic formulae that we consider 'constraints'. The constraint system consists of a signature (a set of first-order function symbols f and predicate symbols r) and a consistent first-order theory Δ. The theory can be given as the set of all sentences valid in a given structure; *e.g.*, of finite trees. The abstract syntax of constraints φ is given below (the symbols $\perp$ and $\top$ stand for the *false* and *true* constants).

$$\text{Constraints} \quad \varphi \ ::= \ r(\bar{x}) \ | \ x = y \ | \ x = f(\bar{y}) \ | \ \perp \ | \ \top \ | \ \varphi_1 \wedge \varphi_2 \ | \ \exists x \varphi$$

The semantics uses the test of satisfiability of constraints ("$\Delta \models \varphi \leftrightarrow \perp$").[8]

For *modeling states*, we use formulae called *configurations* C which are, for now, of the form $\varphi : E$, *i.e.*, composed of a *constraint store* φ (a constraint) and a *continuation* E (an expression) by a symbol ":" (read: "Colon And"). The initial state is modeled as the configuration constituted by the "empty constraint store" $\top$ (standing for the *true* constraint) and the query expression; for example, if the query is `length([a|Z],N), prime(N)?`, as

$$\underbrace{\top}_{\text{constraint store}} \ : \ \underbrace{(x = cons(y,z) \wedge y = a) \,\&\, length(x,n) \,\&\, prime(n)}_{\text{continuation}}$$

Concluding the introduction of syntax, we recapitulate that we have in total three "And" symbols: ":" for constituting configurations, "$\wedge$" to compose constraints, and "&" to compose expressions (which constitute continuations and the body of predicate definitions).

All *state transitions* must be authorized by a reduction rule. Since the first &-conjunct in the continuation is a constraint, we apply the Constraint Elaboration rule to obtain the successor state of the initial state:

$$\underbrace{\top \wedge (x = cons(y,z) \wedge y = a)}_{\text{constraint store}} \ : \ \underbrace{length(x,n) \,\&\, prime(n)}_{\text{continuation}}.$$

Now, the first &-conjunct in the continuation is an atom, and we apply the Unfolding rule.

The first &-conjunct in the continuation being a ;-disjunction, we next apply the Or-Elaboration rule. Then, the configuration modeling the successor state is composed of two configurations by the ";"-symbol. Each of the two has a copy of the constraint store and of the remaining conjuncts in the continuation.

Until now, states were represented by configurations of the form $\varphi : E$, and we could apply the reduction rules on these configurations in the empty context (which is noted "$\bullet$"). Now, the configuration is of the form $C\,;C'$. Here, $\bullet\,;C'$ is the context of C. Hence, we may apply a reduction rule on C; if this application yields C'', then the configuration $C\,;C'$ reduces to the configuration $C''\,;C'$. This means that the reduction of an ";"-disjunction is realized by rewriting the first configuration-disjunct.

[8] In the example, the constraints correspond to the term equations in Prolog. They are interpreted over the logical structure Δ of finite trees. Here, The signature does not contain any predicate symbols r.

Here, since the first $\&$-conjunct in the continuation is a constraint which is incompatible with the constraint store, the first continuation-disjunct reduces, by applying the Constraint-Elaboration rule, to the configuration $\perp$ (the fail-configuration). Hence, we obtain the successor state modeled by a configuration of the form $\perp : C$. Since the Sequential-Or congruence law postulates in particular that $\perp \,;\, C \;\equiv\; C$ ("$\perp$ is a neutral element for "$;$"-composing configuration-disjuncts"), the configuration C models the same state.

In order to illustrate how we model constraint simplification declaratively (instead of using unification), we consider one more transition. Application of the Constraint-Elaboration rule yields the state which is modeled by the configuration:

$$\mathrm{x = cons(y,z) \wedge y = a \wedge x = cons(y,z) \wedge n = succ(m) : length(z,m) \,\&\, prime(n)}$$

and which is modeled also by the following configuration which is congruent according to the Simplification congruence law.[9]

$$\mathrm{(x = cons(y,z) \wedge y = a \wedge n = succ(m)) : length(z,m) \,\&\, prime(n).}$$

To recapitulate and to compare with SLD-resolution, we model *Leftmost-Selection* through the form of the reduction rules and through the Sequential-And congruence laws (which say that an $\&$-conjunction of expressions in a continuation forms an ordered sequence; *i.e.*, the operator "$\&$" is not commutative). We model the *Topmost Selection* rule through the definition of contexts and through the Sequential-Or congruence laws (*i.e.*, "$;$" is not a commutative operator on configuration-disjuncts).[10] We model *Success* by the fact that the continuation of the first configuration-disjunct in a configuration is equal to $\top$. That is, the configuration is of the form $\varphi : \top \,;\, C$. A transition sequence coming to a state modeled by such a configuration terminates in this state.[11] We model *Failure* by the fact that a configuration reduces to the fail-configuration $\perp$ if the first $\&$-conjunct in the continuation is a constraint which is is logically inconsistent with the constraint store. The fail-configuration may be the first configuration-disjunct in a configuration modeling an execution state. Then omitting the $\perp$-disjunct yields a congruent configuration which models the same state. This is how we model *Backtracking*.

[9]The Simplification congruence law does not inflict on the reduction relation (since the Constraint Elaboration rule depends directly on the logical meaning of constraints). It is used for modeling the effect of the constraint solver. It will be used also in Section 6 for modeling the lifetime of local variables.

[10]The Sequential-Or congruence laws are formulated only for configurations. The Or-Elaboration rule implies, however, that one can apply them also to expressions without changing the semantics; *i.e.*, an "$;$"-disjunction of expressions may also written as their ordered sequence.

[11]We model the behavior of Edinburgh-style Prolog where further solutions have to be requested explicitly. Request of further solution amounts to failing the first alternative/configuration.

4 The Cut ("!")

We introduce the semantics defined in Figure 2, for Horn clauses with the addition of the cut operator (and still without the account of "$\exists$").

```
p(t1):- q1.
p(t2):- q21, !, q22, q23.
p(t3):- q3.
p(t4):- q4.
```

We translate the above predicate definition in concrete Prolog syntax with cut into the following one in abstract syntax:

$$p(x) \leftrightarrow (\underbrace{(x = t1 \,\&\, q1)}_{\text{clause before/}} \,;\; !\,(\underbrace{x = t2 \,\&\, q21}_{\text{goals before/}},\; \underbrace{q22 \,\&\, q23}_{\text{after cut}},\; \underbrace{x = t3 \,\&\, q3 \,;\, x = t4 \,\&\, q4)}_{\text{clauses after cut}}\;)$$

which uses expressions of the form $!(E, E_1, E_2)$, *i.e.*, composed of three sub-expressions (E, the cut-guard; E_1 and E_2, the first and second alternatives) by the symbol "!" (read: "cut"). This symbol cannot given a logical meaning which corresponds to its operational semantics.

The translation above gives an idea on how such a translation can be done systematically. It is given for the first time in [13] for replacing cut by DEC-10 Prolog's if-then-else operator (which is implemented unsoundly).

We will now define the operational semantics of cut-expressions. If an execution state is modeled by a configuration where the first conjunct in the continuation is a cut-expression, then the Cut-Elaboration rule is applied.[12] The successor state is modeled in the extended-syntax configurations, namely by a configuration of the form $!(C_1, E, C_2)$ composed of a configuration C_1 (the guard-configuration), an expression E (the guard-continuation) and another configuration C_2 (the alternative configuration) by the "!" symbol.

The definition of contexts yields that in order to model the next successor state, one reduces the guard-configuration; *i.e.*, the reduction of a cut-configuration takes place in the guard-configuration. The guard-configuration can itself reduce into a cut-configuration, and so forth.

In the case where the guard-configuration in a cut-configuration has failed, *i.e.*, the execution state is modeled by a cut-configuration of the form $!(\bot, E, C)$, we apply the Cut-Failure rule to derive the successor state. It is modeled by the alternative configuration C. In the case where the guard-configuration has succeeded, *i.e.*, the execution state is modeled by a cut-configuration of the form $!((\varphi\!:\!\top \,;\, C_1), E, C_2)$, we apply the Cut-Success rule to derive the successor state. It is modeled by $\varphi\!:\!E$, *i.e.*, the first disjunct of the guard-configuration with the guard-continuation. The second disjunct C_1 of the guard-configuration and the alternative configuration C_2 are put away with. This is how we model "pruning parts of the search tree."

[12] To obtain an intuition of the Cut-Elaboration rule, $\varphi : !(E_0, E_1, E_2) \,\&\, E \longrightarrow !((\varphi : E_0), (E_1 \,\&\, E), (\varphi : E_2 \,\&\, E))$, reformulate the Or-Elaboration rule to $\varphi : (E_0 \,\&\, E_1 \,;\, E_2) \,\&\, E \longrightarrow (\varphi\!:\!E_0 \,\&\, E_1 \,\&\, E) \,;\, (\varphi\!:\!E_2 \,\&\, E)$.

SYNTAX

Expressions	$E \ ::= \ \varphi \ \mid \ p\bar{x} \ \mid \ E_1 \,\&\, E_2 \ \mid \ E_1 \,;\, E_2 \ \mid !(E, E_1, E_2)$
Configurations	$C \ ::= \ \varphi{:}E \ \mid \ \bot \ \mid \ C_1 \,;\, C_2 \ \mid !(C_1, E, C_2)$
Contexts	$\mathcal{C} \ ::= \ \bullet \ \mid \ \mathcal{C} \,;\, C \ \mid \ !(\mathcal{C}, E, C)$

REDUCTION RULES

Cut Elaboration	$\varphi{:}!(E_0, E_1, E_2) \,\&\, E \ \longrightarrow \ !((\varphi{:}E_0), (E_1 \,\&\, E), (\varphi{:}E_2 \,\&\, E))$
Cut Failure	$!(\bot, E, C) \ \longrightarrow \ C$
Cut Success	$!((\varphi{:}\top \,;\, C_1), E, C_2) \ \longrightarrow \ \varphi{:}E$

Figure 2: The Cut ("!")

5 Coroutining ("?")

Given the syntax of expressions as in Section 3 (*cf.*, in Figure 1), we will extend it with *conditional expressions*. For now (*cf.*, Figure 3), these are expressions of the form $?(\varphi, E_1, E_2)$ composed of a constraint φ (the guard) and expressions E_1 (the then-expression) and E_2 (the else-expression) by the ternary symbol "?" (read: "cond"). The logical meaning is "if φ then E_1 else E_2" which is "$\varphi \wedge E_1 \ \vee \ \neg\varphi \wedge E_2$." Conditional expressions can be used to express *negated constraints*: $\neg\varphi$ is logically equivalent to $?(\varphi, \bot, \top)$.

As an example, we give a new definition of the *length* predicate (*cf.* the guarded Horn clause $length(x,n) \coloneq x = nil \,\|\, n = zero$), again omitting existential quantification:

$$length(x,n) \leftrightarrow \ ?(x = nil, \ n = zero, \\ x = cons(y,z) \wedge n = succ(m)) \,\&\, length(z,m)$$

The operational semantics of conditional expressions can be described as follows: For each encountered (*i.e.*, elaborated) cond-expression $?(\varphi, E_1, E_2)$, test whether the constraint store entails or disentails φ. If yes, execute E_1 or E_2, respectively ("fire" or "trigger the coroutine"). Otherwise, suspend and repeat the test ("resume") after a modification of the constraint store (*i.e.*, after a constraint elaboration).

The operational semantics outlined above implements the "freeze" operator of Prolog-II. For example, `freeze(X,p(X))` corresponds to $?(x = a, p(x), p(x))$ where a can be any constant.

We will now formalize this operational semantics, so that it: (1) is sound with respect to the logical semantics, (2) reflects the suspension/resumption mechanism, (3) accounts for multiple suspensions, (4) models the trigger condition declaratively, and (5) accounts for the incrementality and locality (to each guard) of the possibly often repeated entailment and disentailment tests.

We introduce a new syntact entity, the *pool*, which becomes another constituent of configurations, besides the constraint store and the contin-

SYNTAX

Expressions	$E \ ::= \ \varphi \mid p\bar{x} \mid E_1 \,\&\, E_2 \mid E_1 \,;\, E_2 \mid ?(\varphi, E_1, E_2)$
Pools	$P \ ::= \ ?(\varphi, E_1, E_2) \mid \top \mid P_1 \wedge P_2$
Configurations	$C \ ::= \ P : \varphi : E \mid \bot \mid C_1 \,;\, C_2$

REDUCTION RULES

Cond Elaboration	$P : \varphi : ?(\varphi_0, E_1, E_2) \,\&\, E \longrightarrow P \wedge ?(\varphi_0, E_1, E_2) : \varphi : E$
Cond Success	$P \wedge ?(\top, E_1, E_2) : \varphi : E \longrightarrow P : \varphi : E_1 \,\&\, E$
Cond Failure	$P \wedge ?(\bot, E_1, E_2) : \varphi : E \longrightarrow P : \varphi : E_2 \,\&\, E$
Control	Cond Success and Cond Failure have priority.

CONGRUENCE LAWS

Parallel And	$(\{Pools\}, \ \wedge, \ \top)$ is a commutative monoid.
Relative Simp.lification	$?(\varphi_1, E_1, E_2) : \varphi : E \ \equiv \ ?(\varphi_2, E_1, E_2) : \varphi : E$ if $\Delta \models \varphi_1 \wedge \varphi \leftrightarrow \varphi_2 \wedge \varphi$.

Figure 3: Coroutining ("?")

uation. The pool of a configuration modeling some state describes which cond-expressions have been encountered (*i.e.*, elaborated) but not yet fired so far. Thanks to the Paralled-And congruence laws, these cond-expressions form an unordered list, *i.e.*, a multiset. This is important because this models the fact that the tests resume on all suspensions at the same time, and there is no *a priori* order on the coroutines that can be fired.

The initial state of an execution is now modeled by a configuration with an empty pool, *i.e.*, equal to $\top$ (and with an empty constraint store, as before).

In a first version, the three reduction rules for cond-expressions are the following.

Cond Elab.	$P : \varphi : ?(\varphi_1, E_1, E_2) \,\&\, E \longrightarrow P \wedge ?(\varphi_1, E_1, E_2) : \varphi : E$
Cond Success	$P \wedge ?(\varphi_1, E_1, E_2) : \varphi : E \longrightarrow P : \varphi : E_1 \,\&\, E$ if $\Delta \models \varphi \rightarrow \varphi_1$
Cond Failure	$P \wedge ?(\varphi_1, E_1, E_2) : \varphi : E \longrightarrow P : \varphi : E_2 \,\&\, E$ if $\Delta \models \varphi \rightarrow \neg\varphi_1$

In order to reflect that the resumption of the tests happens after each modification of the constraint store and the coroutine is triggered as soon as possible, we add an applicability condition to all reduction rules other than Cond Success or Failure. This condition says that neither the Success nor the Cond Failure rule are applicable. The condition can, of course, also be formulated locally to each rule. Namely, as: The configuration to be reduced is not congruent to a configuration of the form $P \wedge ?(\varphi_1, \_, \_) : \varphi : \_$ where $\Delta \models \varphi \rightarrow \varphi_1$ or $\Delta \models \varphi \rightarrow \neg\varphi_1$.

The semantics given so far satisfies the first four points raised above, but not point (5). The questions open are: Can the trigger condition (*i.e.*, entailment or disentailment) be tested locally, *i.e.*, by looking only at the particular cond-expression? Can this test be done incrementally?

From now on, we will assume that the constraint system can implement *Relative Simplification* [2], which simplifies a constraint φ_1 relative to a constraint φ to a constraint φ_2 such that (1) the conjunction of φ with φ_1 is equivalent to the conjunction with φ_2, (2) if φ entails $\exists \bar{x} \varphi_1$, then $\exists \bar{x} \varphi_2$ is equivalent to *true*,[13] and (3) if φ disentails φ_1, then φ_2 is the *false* constraint; *i.e.*,

$$\begin{aligned} &\Delta \models \varphi \wedge \varphi_1 \leftrightarrow \varphi \wedge \varphi_2 \ ; \\ &\Delta \models \varphi \rightarrow \exists \bar{x} \varphi_1 \ \text{ iff } \ \Delta \models \exists \bar{x} \varphi_2 \leftrightarrow \top \ ; \\ &\Delta \models \varphi \rightarrow \neg \exists \bar{x} \varphi_1 \ \text{ iff } \ \varphi_2 = \bot \ . \end{aligned}$$

Relative Simplification yields incremental and local, sound and complete tests of entailment and disentailment.

We now add Relative Simplification congruence law and replace the Cond Success and Failure rules by the new versions given in Figure 3. This completes modeling the execution of coroutines.

To come back to the point raised in Section 3 on the difference between $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \,\&\, \varphi_2$, we can now construct an example and use the formal semantics to show that in the one case a suspended cond-expression is fired (possibly leading to an infinite loop) and in the other it is not. Take the definition $p(x) \leftrightarrow ?(x = f(y,a), p(x), \bot)$ and the initial states $\top : \top : p(x) \,\&\, (x = f(y,z) \wedge z = b)$ and $\top : \top : p(x) \,\&\, x = f(y,z) \,\&\, z = b$ (corresponding to the queries `p(X), X=f(Y,b)` and `p(X), X=f(Y,Z), Z=b`, respectively).

To indicate a source of *non-confluence*, one can construct an example where the order chosen for the firing of two competing suspended cond-expressions yields two different behaviors (*e.g.*, one leading to an infinite loop and the other not). This is due to the fact that, in our semantics, both cond-expressions are moved from the ("concurrent") pool to the ("sequential") continuation where they are ordered by their composition with "&".[14] Take the definitions $p(x) \leftrightarrow ?(x = a, p(x), \bot)$ and $q(x) \leftrightarrow ?(x = a, \bot, \bot)$ and the initial state $\top : \top : p(x) \,\&\, q(x) \,\&\, x = a$. Execution may lead to either of the states $\top : (x = a) : \bot \,\&\, p(x)$ or $\top : (x = a) : p(x) \,\&\, \bot$.

We will discuss the interaction of cut and coroutining in Section 7.

6 Existential quantification ("$\exists$")

The logically correct version of the definition of the *length* predicate in Section 3 (when $\leftrightarrow$ means "if and only if") has to quantify the variable y in a

[13] As a special case: If φ entails φ_1, then φ_2 is equivalent to *true*.

[14] One may, of course, design a semantics with a different, more fair execution strategy for coroutines.

constraint and the variables z and m in an expression:

$$\begin{aligned} length(x,n) \leftrightarrow\ & (x = nil \wedge n = zero)\,; \\ & (\exists z \exists m\, \underbrace{(\exists y (x = cons(y,z)) \wedge n = succ(m) \,\&\, length(z,m))}_{\text{expression being quantified}})) \end{aligned}$$

The previous version of the Unfolding rule tacitly assumes that the program always contains an appropriate version of the definition of the predicate being unfolded. We did not model that such a version is obtained, for each unfolding, by consistent renaming of all variables occurring in the definition. Also, in the semantics present so far, the new variables introduced by the unfolding are visible everywhere and never disappear again. This is not a complete modeling of the actual operational semantics.

From now on, having extended the syntax of expressions with "$\exists$" (*cf.*, Figure 4), we may assume that the expression defining a predicate $p\bar{x}$ does not have any free variables other than the ones in the tuple $\bar{x}$. Hence, the new form of the Unfolding rule does not add any free variables to the configuration to which it is applied. Its applicability condition $\bar{x} \cap \bar{y} = \emptyset$ (to avoid capturing between the actual parameters $\bar{x}$ and the formal ones $\bar{y}$) can be satisfied (with the Quantifier Renaming congruence law) if $\bar{x}$ is in the scope of an existential quantifier; hence, it is sufficient to require that the free variables in the query (initial configuration) do not occur in the program.

A quantified expression cannot be elaborated directly. The scope of the quantified variables has to be extended first (by application of the Quantifier Elaboration law). This is possible only if they can not be captured, *i.e.*, come into the scope of a different quantifier. In order to extend the scope of variables that have the same name as others, they have to be renamed first (by application of the Quantifier Renaming law). After the scope of the quantified variables has been extended ("the quantifier is pulled out"), *i.e.*, the whole configuration instead of just the expression is quantified, the configuration can be reduced thanks to the definition of contexts.

Finally, the scope of a variable in an expression can be reduced to any sub-expression that contains all its occurrences ("the quantifier is pulled in"). This could be modeled by congruence laws of the form $\exists x(T_1 \circ T_2) \equiv (\exists x\ T_1) \circ T_2$ where $x \notin \mathcal{V}(T_2)$, for any trees T_1 and T_2 of any syntactic entity and any composition symbol $\circ$. In fact, the Quantifier Mobility law in the version given here suffices for our semantics, together with the Quantifier Exchange law. The Simplification law then takes care of removing quantifiers of redundant ("auxiliary") variables (such as in $\exists x\ x = nil \wedge \varphi$ where $x \notin \mathcal{V}(\varphi)$) and of removing variables which do no longer appear in a configuration, since $\exists x \varphi \equiv \varphi$ if $x \notin \mathcal{V}(\varphi)$.

This is how we account for the introduction of variables as well as for their scope (*i.e.*, their visibility and life-time), namely through congruence laws which reflect laws for logical quantification.

SYNTAX

Expressions	$E \ ::= \ \varphi \mid p\bar{x} \mid E_1 \,\&\, E_2 \mid E_1\,;\,E_2 \mid \exists x E$
Configurations	$C \ ::= \ \varphi\!:\!E \mid \bot \mid C_1\,;\,C_2 \mid \exists x\, C$
Contexts	$\mathcal{C} \ ::= \ \bullet \mid \mathcal{C}\,;\,C \mid \exists x\, \mathcal{C}$

CONGRUENCE LAWS

Quantifier Exchange	$\exists x \exists y E \ \equiv\ \exists y \exists x E, \quad \exists x \exists y C \ \equiv\ \exists y \exists x C$
Quantifier Renaming	$C_1 \ \equiv\ C_2$ if C_1 and C_2 equal up to α-renaming
Quantifier Mobility	$\exists x(\varphi\!:\!E\,;\,C) \ \equiv\ (\exists x \varphi)\!:\!E\,;\,\exists x C$ if $x \notin \mathcal{V}(E)$

Figure 4: Existential quantification ("$\exists$")

7 The complete semantics

The addition of the features discussed in the previous three sections to the base case discussed in Section 3 yields the semantics defined in Figure 5. Two of these feature combinations need to be discussed. We leave the discussion of the combination of "$\exists$" and "?" to the full version of this paper.

Cut and Coroutining The complete semantics accounts in particular for the interaction between cut- and cond-expressions. Here, we must remove some ambiguities that are often left open in language specifications or manuals. There are two interesting cases, namely (1) where a suspension lies in the scope of a cut, and (2) where a suspension calls a cut upon triggering.

Our formalization of the semantics defines (*i.e.*, specifies through the form of the Cut Success and Failure rules) that a cut ignores suspensions for its decision. More precisely, if a cut is reached at a moment of execution where a coroutine is suspending, then a potential failure of this co-routine is not anticipated but the cut is executed as if the co-routine did not exist. (It is crucial, of course, that this coroutine can indeed not be fired; this is assured by the additional applicability condition on all rules other than Cond Success and Failure expressing that these two rules have priority.)

In the second case it seems better to leave the specification open and admit non-confluence of terminating executions.[15] We recall that we postulate that "$\wedge$" is a commutative composition symbol over cond-expressions in the Paralled-And congruence law, and thus do not fix the order of reactivations of competing suspensions. In the case where a reactivation leads to the call of an order-dependent control-operator such as the cut, its decision and the subsequent program execution are no longer determined before-hand. One can easily construct an example where two coroutines are fired one after the other and where one of the two calls a cut whose decision is influenced by the other. Take the definitions $p(x,y) \leftrightarrow ?(x=a, y=b, \bot)$ and

[15] At the end of Section 5 we mentioned non-confluence even in absence of the cut.

$q(X,Y) \leftrightarrow ?(x=a, !(r(Y),\top,\bot),\bot)$ and $r(y) \leftrightarrow (y=a; y=b)$ and the initial state $\top : \top : p(x,y) \,\&\, q(x,y) \,\&\, x=a$. Its execution leads to either of the two states $\top : x=a, y=b : !(r(y),\top,\bot)$ or $\top : x=a : !(r(y),\top \,\&\, y=b,\bot)$ and from there to $\top : x=a, y=b : \top$ or to $\bot$, respectively. We model this *indeterminism* by the fact that our abstract-tree-rewriting system is non-confluent when such a case is admitted.

8 Conclusion and Future Work

We have presented a formal definition of the operational semantics of constraint logic programs with coroutining. What is it good for? We will list a few benefits.

(1) By modeling program execution through reduction rules and through congruence laws, we have been able to state what the (primarily) operational and what the declarative aspects of the languages are. For instance, we have accounted for the fact that the sequences of goals are ordered, but the representations of constraints as conjunctions are not.

(2) We have abstracted away from details that are relevant only for language implementations, and not for the program execution. Clearly, backtracking is merely a matter of space-efficient implementation.

(3) We have modeled the control operators of the program (such as the order statement for goals and clauses and the cut). This enables the programmer to predict the control flow of his program and to evaluate its performance (assuming a complexity bound for the constraint solver, *e.g.*, a quasi-linear one in the case of Prolog-II).

(4) We have been able to make precise the critical interaction between sequential execution (of expressions in the continuation) and concurrent execution (of suspended conditionals in the pool), in particular in the two cases where a suspension lies in the scope of a cut or a suspension calls a cut upon triggering.

As for future work: Not all of the mathematical formulae which model execution states have a first-order logic reading, nor do all of the rewriting steps which model state transitions. It is now possible, however, to investigate which of them do. This can be useful in order to determine which of the programs do have a declarative reading (in first-order logic).

Along the lines of this paper, one can now start modeling other language functionalities which are considered impure such as assert/retract, request of further solutions, backtrackable assignment, cells, and so on.

One interesting branch of future work would consist of investigating confluence proofs for the reduction relation with coroutining (without coroutining, all reduction steps are deterministic), possibly building on work in [12]. As the examples given above show, only "'weak" confluence may possibly hold (*i.e.*, confluence of terminating executions), and even this only under a restriction on the use of cut.

Acknowledgements We thank Harald Ganzinger, Michael Hanus, Sverker Janson, Olivier Ridoux and Peter Van Roy for discussions and com-

SYNTAX

Expressions	$E ::= \varphi \mid p\bar{x} \mid E_1 \& E_2 \mid E_1 \,;\, E_2 \mid !(E, E_1, E_2) \mid ?(\bar{x}, \varphi, E_1, E_2) \mid \exists x E$
Pools	$P ::= ?(\bar{x}, \varphi, E_1, E_2) \mid \top \mid P_1 \wedge P_2$
Configurations	$C ::= P : \varphi : E \mid \bot \mid C_1 \,;\, C_2 \mid \exists x C \mid !(C_1, E, C_2)$
Contexts	$\mathcal{C} ::= \bullet \mid \mathcal{C} \,;\, C \mid !(\mathcal{C}, E, C) \mid \exists x\, \mathcal{C}$

REDUCTION RULES

Application Elab.	$P : \varphi : p\bar{x} \,\&\, E \;\longrightarrow\; P : \varphi : \exists \bar{y}(\bar{x} = \bar{y} \,\&\, E_1) \,\&\, E$ if $\mathrm{p}\bar{\mathrm{y}} \leftrightarrow \mathrm{E}_1$ in program, $\bar{\mathrm{x}} \cap \bar{\mathrm{y}} = \emptyset$
Constraint Elab.	$P : \varphi : \varphi_1 \,\&\, E \;\longrightarrow P : \varphi \wedge \varphi_1 : E$ if $\Delta \not\models \varphi \wedge \varphi_1 \leftrightarrow \bot,\ \varphi_1 \neq \top$ $\longrightarrow \bot$ if $\Delta \models \varphi \wedge \varphi_1 \leftrightarrow \bot$
Or Elaboration	$P : \varphi : (E_1 \,;\, E_2) \,\&\, E \;\longrightarrow\; (P : \varphi : E_1 \,\&\, E) \,;\, (P : \varphi : E_2 \,\&\, E)$
Cut Elaboration	$P : \varphi : !(E_0, E_1, E_2) \,\&\, E \;\longrightarrow !(P : \varphi : E_0, E_1 \,\&\, E, P : \varphi : E_2 \,\&\, E)$
Cut Failure	$!(\bot, E, C) \;\longrightarrow\; C$
Cut Success	$!((P : \varphi : \top \,;\, C_1), E, C_2) \;\longrightarrow\; P : \varphi : E$
Cond Elab.	$P : \varphi : ?(\bar{x}, \varphi_0, E_1, E_2) \,\&\, E \;\longrightarrow\; P \wedge ?(\bar{x}, \varphi_0, E_1, E_2) : \varphi : E$
Cond Success	$P \wedge ?(\bar{x}, \varphi_1, E_1, E_2) : \varphi : E \;\longrightarrow\; P : \varphi : \exists \bar{x}(\varphi_1 \,\&\, E_1) \,\&\, E$ if $\Delta \models \exists \bar{x} \varphi_1$
Cond Failure	$P \wedge ?(\bar{x}, \bot, E_1, E_2) : \varphi : E \;\longrightarrow\; P : \varphi : E_2 \,\&\, E$
Control	Cond Success and Cond Failure have priority.

CONGRUENCE LAWS

Sequential And	$(\{\mathit{Expressions}\},\ \&\,,\ \top)$ is a monoid.
Sequential Or	$(\{\mathit{Configurations}\},\ ;\,,\ \bot)$ is a monoid.
Parallel And	$(\{\mathit{Pools}\},\ \wedge,\ \top)$ is a commutative monoid.
Simplification	$\varphi_1 \;\equiv\; \varphi_2$ if $\Delta \models \varphi_1 \leftrightarrow \varphi_2$.
Relative Simp.	$?(\bar{x}, \varphi_1, E_1, E_2) : \varphi : E \;\equiv\; ?(\bar{x}, \varphi_2, E_1, E_2) : \varphi : E$ if $\Delta \models \varphi_1 \wedge \varphi \leftrightarrow \varphi_2 \wedge \varphi$.
Quantifier Elab.	$P : \varphi : \exists x E_1 \,\&\, E_2 \;\equiv\; \exists x(P : \varphi : E_1 \,\&\, E_2)$ if $x \notin \mathcal{V}(P : \varphi : E_2)$
Q. Exchange	$\exists x \exists y E \;\equiv\; \exists y \exists x E, \quad \exists x \exists y C \;\equiv\; \exists y \exists x C$
Q. Renaming	$C_1 \;\equiv\; C_2$ if C_1 and C_2 equal up to α-renaming.
Q. Mobility	$\exists x(P : \varphi : E \,;\, C) \;\equiv\; P : \exists x \varphi : E \,;\, \exists x C$ if $x \notin \mathcal{V}(P) \cup \mathcal{V}(E)$

Figure 5: The Complete Semantics

ments.

References

[1] H. Aït-Kaci, B. Dumant, R. Meyer, A. Podelski and P. Van Roy. *Wild LIFE: A User Manual.* Digital Equipment Corporation, Paris Research Laboratory. Rueil-Malmaison, France, April 1993.

[2] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994. (Also appeared in *Fifth Generation Computing Systems*, Tokyo, 1992.)

[3] E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, LNCS 533, pages 67–79, 1991.

[4] M. Carlsson, J. Widèn, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog User's Manual.* SICS, Box 1263, 164 28 Kista, Sweden, 1991.

[5] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2:4, pages 255–292, 1983.

[6] J. Jaffar, M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming* 19 & 20, 503–581, 1994.

[7] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[8] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Fourth ICLP*, pages 858–876. MIT Press, 1987.

[9] R. Milner. Functions as Processes. In *Journal of Mathematical Structures in Computer Science* 2:2, pages 119–141, 1992.

[10] L. Naish. *MU-Prolog 3.1db Reference Manual.* Computer Science Department, University of Melbourne, Melbourne, Australia, May 1984.

[11] L. Naish. Automating Control for Logic Programs. In *JLP* 2:3, 1985, pages 167–184, 1985.

[12] J. Niehren and G. Smolka. A Confluent Calculus for Higher-order Relational Programming. In J-P. Jounnaud, editor, *1st International Conference on Constraints in Computational Logics*, LNCS 845, pages 89–104, 1994.

[13] R. A. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Symposium on Logic Programming*, pages 68–72, The Computer Society Press, 1985.

[14] G. Smolka. Residuation and guarded rules for constraint logic programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 405–419, MIT Press, 1993.

[15] G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research report, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-W-6600 Saarbrücken, Germany, 1993.

[16] G. Smolka. A foundation for higher-order concurrent constraint programming, In J-P. Jounnaud, editor, *1st International Conference on Constraints in Computational Logics*, LNCS 845, pages 50–72, 1994.