A Confluent Calculus for Concurrent Constraint Programming with Guarded Choice

Kim Marriott

Monash University
Clayton 3168, Victoria, Australia
marriott@cs.monash.edu.au

Martin Odersky

Universität Karlsruhe
76128 Karlsruhe, Germany
odersky@ira.uka.de

In Proc. 1st Conf. on Principles and Practice of Constraint Programming, Cassis, France, September 1995.

Abstract. Confluence is an important and desirable property as it allows the program to be understood by considering any desired scheduling rule, rather than having to consider all possible schedulings. Unfortunately, the usual operational semantics for concurrent constraint programs is not confluent as different process schedulings give rise to different sets of possible outcomes. We show that it is possible to give a natural confluent calculus for concurrent constraint programs, if the syntactic domain is extended by a blind choice operator and a special constant standing for a discarded branch. This has application to program analysis.

1 Introduction

Concurrent constraint programming (ccp) [16, 15] is a recent programming paradigm which elegantly combines logical concepts and concurrency mechanisms. The computational model of ccp is based on the notion of a *constraint system*, which consists of a set of constraints and an *entailment* relation. Processes interact through a common *store*. Communication is achieved by *telling* (adding) a given constraint to the store, and by *asking* (checking whether the store entails) a given constraint. Standard ccp provides a non-deterministic guarded choice operator. In the operational semantics of ccp, non-determinism arises in two different ways. First, if the guards of two branches in a committed choice construct are both entailed by the store either branch can be picked. Second, different process schedulings (that is, interleavings of transitions) can lead to different results since a given process scheduling can prune the decision space by selecting a branch in a committed choice before strengthening the store. In this way, some branches that would be entailed by the stronger store might be excluded by the weaker one. This second source of non-determinism means that to find the possible outcomes of a program all process schedulings must be considered in the operational semantics. This need to consider all process schedulings also holds for the denotational semantics of ccp, which expresses parallel composition by interleaving.

Because of the combinatorial explosion of reduction sequences, an interleaving semantics makes reasoning about possible evaluations cumbersome. Yet such reasoning is necessary for many tasks in program analysis, verification and transformation. This contrasts to the situation in both the lambda calculus and (idealised) Prolog. The semantics for both have confluence properties that make it unnecessary to consider different process schedulings. In the lambda calculus, confluence is embodied in the Church-Rosser theorem [1], which says that different reduction sequences starting from the same term can always be re-joined in a common reduct. As a consequence, evaluation in the lambda calculus is deterministic. In Prolog, confluence is embodied in the Switching Lemma [10], which ensures that different literal selection strategies give rise to the same set of answers.

In the context of concurrency, confluence is an even more desirable property since concurrent programs are notoriously difficult to reason about and to analyse. Unfortunately, as we have seen, despite monotonicity of communication, the standard operational semantics for ccp languages is not confluent in the sense that different process schedulings can give rise to different outcomes. This is because of the guarded choice. Indeed, it has become part of the programming language folklore that it is impossible to have both guarded choice and confluence.

We present here a calculus for ccp that is equivalent to ccp's standard semantics in that both lead to the same observations, yet is confluent. Actually we give a calculus for a slightly larger language, ccp_{+0}, which extends ccp by providing a *blind choice* construct and a *failure* constant $\mathbf{0}$. The main difference between our calculus for ccp_{+0} and the standard operational semantics for ccp lies in the treatment of guarded choice. In ccp, once a choice is made, all other alternatives of a choice construct are discarded. In ccp_{+0}, the other alternatives are kept around, but extended with a guarded branch which reduces to $\mathbf{0}$ on termination, indicating that this alternative is only valid if another branch in the guard does not suspend. The calculus distinguishes between the two forms of non-determinism in ccp. Non-determinism arising from multiple guards being enabled is expressed by the blind choice operator in the term language. Process scheduling non-determinism is reflected by a choice among different reduction sequences, analogous to the situation in the lambda calculus. Our main result is a confluence theorem for this calculus, which essentially says that the choice of process scheduling has no influence on the observable behaviour. This is equivalent to the Church-Rosser theorem for the lambda calculus or the Switching Lemma for Prolog. Our result thus refutes the folklore that is impossible to have both guarded choice and confluence. Monotonicity of communication is crucial to our result.

Besides its theoretical interest, our confluent calculus has practical applications in static analysis of ccp. Lack of confluence in the usual operational semantics and denotational semantics means that program analysis cannot be directly based on these semantics, as the cost of considering all process schedulings in an analysis is prohibitive. There have been two main approaches to overcome this difficulty. The first is to use a fixed process scheduling, but then to "re-execute" the program until a fixpoint is reached. This was suggested in [4] for concurrent logic programs and extended in [5] to ccp. This may be expensive and is inherently imprecise because re-execution confuses the behaviour of different branches. The second approach is to give a non-standard operational semantics for ccp which is confluent but which approximates the usual ccp operational semantics by allowing more reductions. Analyses are then proved correct with respect to this approximate operational semantics. This was suggested in [2, 3] for concurrent logic programs and couched in [17, 6] in the slightly different context of ccp as a transformation from a program written in full ccp to an approximating program written in a subset of ccp for which the usual operational semantics is confluent. The disadvantage of this approach is an inherent loss of precision in the analysis because of the approximation introduced in the new semantics or in the program transformation. Our calculus, we believe therefore, provides a better basis for analysis for two reasons. First, because the calculus is confluent, there is no need to introduce complex artificial semantics or transformations as efficient analysis can be directly based on the calculus. Second, because the calculus gives the same observational behaviour as the usual operational semantics, there is no inherent loss of precision and the analysis can be more accurate.

Our result showing that the ccp_{+0} programs are confluent generalizes confluence results of Maher [11] and Saraswat et al [15] about deterministic ccp subsets and Falaschi et al [6] identification of subclasses of ccp for which the usual operational semantics is confluent. Montanari et al [12] give a confluent operational semantics for a variant of ccp with both indeterminism (blind choice) and nondeterminism (angelic choice), however they do not consider guarded choice. Niehren and Smolka have introduced the δ [13] and ρ [14] calculi which have strong connections to the π-calculus and deterministic ccp respectively. They have shown that both of these calculi are confluent. However, unlike our calculus neither the ρ nor the δ calculus has a non-deterministic guarded choice operator.

The rest of this paper is organized as follows. Section 2 introduces the standard operational semantics of the ccp languages. Section 3 presents our calculus. Section 4 shows that reduction in our calculus is confluent and Section 5 shows that the calculus and operational semantics of ccp are observationally equivalent. Section 6 sketches an application of our calculus to the analysis of ccp programs. Section 7 concludes.

2 Concurrent Constraint Programming

Concurrent constraint programming was proposed by Saraswat [16, 15]. We follow here the definition given in [15], which is based on the notion of cylindric constraint system.

A *cylindric constraint system* [7] is a structure $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \exists \rangle$ such that:

1. $\langle \mathcal{C}, \leq \rangle$ is a complete algebraic lattice, where $\sqcup$ is the lub operation (representing logical and), and *true*, *false* are the least and the greatest elements

of $\mathcal{C}$, respectively;

2. For each $x \in Vars$ the function $\exists_x : \mathcal{C} \to \mathcal{C}$ is a *cylindrification operator*:
   (E1) $\exists_x c \leq c$,
   (E2) $c \leq c'$ implies $\exists_x c \leq \exists_x c'$,
   (E3) $\exists_x(c \sqcup \exists_x c') = \exists_x c \sqcup \exists_x c'$,
   (E4) $\exists_x \exists_y c = \exists_y \exists_x c$;
3. For each $x, y \in Vars$, $\mathcal{C}$ contains the *diagonal element*, d_{xy}, which satisfies:
   (D1) $d_{xx} \leq true$,
   (D2) if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
   (D3) if $x \neq y$ then $c \leq d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$.

As usual, we take $c = c'$ iff $c \leq c' \ \wedge \ c' \leq c$. The cylindrification operators essentially model existential quantification and so are useful for defining a hiding operator in the language. Note that if $\mathbf{C}$ models the equality theory, then the diagonal element d_{xy} can be thought of as the formula $x = y$.

Deviating slightly from the treatment of [15], we will base our exposition of ccp on renamings instead of diagonal elements. Renamings can be defined in terms of diagonal elements as follows.

Definition. Let x and y be variables and let $c \in \mathbf{C}$. Then the *renaming* $[y/x]c$ of y for x in c is the constraint $\exists_x(d_{xy} \sqcup c)$.

Definition. The *free variables* $\mathrm{fv}(c)$ of $c \in \mathbf{C}$ is the set $\{x \mid \exists_x c \neq c\}$.

The following proposition shows that we can consistently rename the free variables of a constraint.

Proposition 2.1 Let $c \in \mathbf{C}$ and let x and y be variables such that $y \notin \mathrm{fv}(c)$. Then $\exists_y [y/x]c = \exists_x c$.

The description and semantics of the ccp class of languages is parametric with respect to an underlying cylindric constraint system $\mathbf{C}$. The syntax of agents M and programs P is given by the grammar:

(Agent)	$M ::= c \mid R \mid p\overline{y} \mid M \cdot M \mid \exists_x M$
(Choice)	$R ::= R \mathbin{[\!]} R \mid c \mapsto M$
(Program)	$P ::= D \,;\, M$
(Declarations)	$D ::= D,\, D \mid p\overline{x} := M$

Two fundamental agents are the *tell* operation c which adds the constraint c to the store and the guarded choice among *ask* operations $[\!]_{i=1}^{n} c_i \mapsto M_i$ which evaluates some M_i, provided the corresponding *guard* c_i is entailed by the store. An agent can also be a *procedure call* $p\overline{y}$, where $\overline{y}$ is a vector of parameters $(y_1, \ldots, y_n)$. We assume that every procedure identifier p has exactly one declaration of the form $p(x_1, \ldots, x_n) := M$ in a program and that the lengths of actual and formal argument lists match. Agents can be combined using parallel composition $(\cdot)$. The quantifier $\exists_x M$ hides the use of variable x inside the agent M. We will often use the word *term* as a synonym for *agent*.

$$\textbf{R1}\ \langle c, d\rangle \xrightarrow{ccp} \langle true, c \sqcup d\rangle \qquad \text{where } c \neq true$$

$$\textbf{R2}\ \langle \textstyle\bigsqcap_{i=1}^{n} c_i \mapsto M_i, d\rangle \xrightarrow{ccp} \langle M_j, d\rangle \text{ where } j \in [1, n] \text{ and } c_j \leq d$$

$$\textbf{R3}\ \frac{\langle M, c\rangle \xrightarrow{ccp} \langle M', c'\rangle}{\begin{array}{c}\langle M \cdot N, c\rangle \xrightarrow{ccp} \langle M' \cdot N, c'\rangle \\ \langle N \cdot M, c\rangle \xrightarrow{ccp} \langle N \cdot M', c'\rangle\end{array}}$$

$$\textbf{R4}\ \frac{\langle M, d \sqcup \exists_x c\rangle \xrightarrow{ccp} \langle N, d'\rangle}{\langle \exists_x^d M, c\rangle \xrightarrow{ccp} \langle \exists_x^{d'} N, c \sqcup \exists_x d'\rangle}$$

$$\textbf{R5}\ \langle p\overline{y}, c\rangle \xrightarrow{ccp} \langle [\overline{y}/\overline{x}]M, c\rangle \qquad \text{where } (p\overline{x} := M) \in D$$

Fig. 1. The transition system T_D.

Free variables $\mathrm{fv}(M)$ and *renamings* $[x/y]M$ have their usual inductive definitions, where the cases where M is a constraint are as defined previously. Following the usual convention for reduction systems, we identify α-renamable terms. That is, $\exists_x M$ and $\exists_x [y/x]M$ are regarded as the same term, provided that $y \notin \mathrm{fv}M$. Proposition 2.1 shows that this identification is consistent with our definition of a constraint system.

The standard operational model of ccp is given as a transition system over *configurations*. A configuration consists of a ccp agent and a constraint representing the current store. The transition system T_D is specified with respect to a set of procedure declarations D. Figure 1 gives the rules in the transition system. Constraints are added to the store (R1). A guarded choice is reduced non-deterministically by choosing a branch whose guard is enabled (R2). (R3) describes parallelism as interleaving. To describe locality (R4) the syntax of existentially quantified agents is extended by allowing agents of the form $\exists_x^d M$. This represents an agent in which x is local to M and d is the "hidden" store that has been produced locally by M on x. Initially the local store is empty, that is, $\exists_x M = \exists_x^{true} M$. The execution of a procedure call is modelled by (R5). We write $\xrightarrow{ccp}\!\!\!\!\rightarrow$ for the reflexive and transitive closure of $\xrightarrow{ccp}$.

The standard observable behavior of a ccp agent is the set of possible constraint stores which can result when the agent is reduced to a normal form. A configuration S is in *normal form* if it cannot be reduced further. Infinite reduction sequences are equated to the constraint *false*.

Definition. Let P be the ccp program D ; M. Then $P \Downarrow_{ccp} c$ if there is a normal form $\langle N, c\rangle$ such that $\langle M, true\rangle \xrightarrow{ccp}\!\!\!\!\rightarrow \langle N, c\rangle$ in the transition system T_D. *P diverges*, written $P \Uparrow_{ccp}$ iff there is an infinite T_D-transition sequence starting with $\langle M, true\rangle$.

Definition. The set of *observations* of a program P, $Obs(\xrightarrow{ccp}, P)$ is

$$\{c \mid M \Downarrow_{ccp} c\} \cup \{\mathit{false} \mid M \Uparrow_{ccp}\}.$$

Example 2.2 The following declaration D defines an agent $merge$, which non-deterministically merges its two input streams x and y into an output stream z. The constraint domain is equations over finite terms. We use $[]$ to denote the empty stream, and $[u \mid v]$ to denote the stream with head u and tail v.

$$\begin{array}{l}
merge(x, y, z) := \\
\qquad \exists_{x'}\exists_u\ x = [u \mid x'] \mapsto \exists_{x'}\exists_u\exists_{z'}(x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x', y, z')) \\
\quad [\!] \ \exists_{y'}\exists_u\ y = [u \mid y'] \mapsto \exists_{y'}\exists_u\exists_{z'}\ (y = [u \mid y'] \cdot z = [u \mid z'] \cdot merge(x, y', z')) \\
\quad [\!] \ x = [] \mapsto z = y \\
\quad [\!] \ y = [] \mapsto z = x.
\end{array}$$

Let P be the program D ; $x = [a] \cdot merge(x, y, z) \cdot y = [b]$. A reduction sequence using left-most agent scheduling is:

$$\begin{array}{ll}
 & \langle x = [a] \cdot merge(x, y, z) \cdot y = [b], true\rangle \\
(R1) \xrightarrow{ccp} & \langle merge(x, y, z) \cdot y = [b], x = [a]\rangle \\
(R5) \xrightarrow{ccp} & \langle M \cdot y = [b], x = [a]\rangle \\
(R2) \xrightarrow{ccp} & \langle \exists_{x'}\exists_u\exists_{z'}\ (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x', y, z')) \cdot y = [b], x = [a]\rangle \\
(R1) \xrightarrow{ccp} & \langle \exists_{x'}^{x'=[]}\exists_u^{u=a}\exists_{z'}\ z = [u \mid z'] \cdot merge(x', y, z')) \cdot y = [b], x = [a]\rangle \\
(R1) \xrightarrow{ccp} & \langle \exists_{x'}^{x'=[]}\exists_u^{u=a}\exists_{z'}\ merge(x', y, z') \cdot y = [b], x = [a] \sqcup \exists_{z'} z = [a \mid z']\rangle \\
(R5) \xrightarrow{ccp} & \langle \exists_{x'}^{x'=[]}\exists_u^{u=a}\exists_{z'}\ M' \cdot y = [b], x = [a] \sqcup \exists_{z'} z = [a \mid z']\rangle \\
(R2) \xrightarrow{ccp} & \langle \exists_{x'}^{x'=[]}\exists_u^{u=a}\exists_{z'}\ y = z' \cdot y = [b], x = [a] \sqcup \exists_{z'} z = [a \mid z']\rangle \\
(R1) \xrightarrow{ccp} & \langle true \cdot y = [b], x = [a] \sqcup z = [a \mid y]\rangle \\
(R1) \xrightarrow{ccp} & \langle true \cdot true, y = [b] \sqcup x = [a] \sqcup z = [a, b]\rangle
\end{array}$$

where M and M' are appropriate renamings of the definition of $merge(x, y, z)$ and $merge(x', y, z')$ respectively. This reduction sequence gives the observable behavior $y = [b] \sqcup x = [a] \sqcup z = [a, b]$.

In fact this is the only reduction sequence possible with a leftmost agent scheduling. With rightmost agent scheduling, however, the only observation is $y = [b] \sqcup x = [a] \sqcup z = [b, a]$. Thus

$$Obs(\xrightarrow{ccp}, P) \supseteq \{y = [b] \sqcup x = [a] \sqcup z = [b, a], y = [b] \sqcup x = [a] \sqcup z = [a, b]\}.$$

In fact, examination of the (large number of) other agent schedulings shows that these are the only observable behaviours. A more efficient way to show that these are the only observable behaviours will be discussed in the next section.

This example clearly shows the non-confluence of the standard operational semantics, as different agent schedulings give different results.

3 The Concurrent Constraint Calculus

In this section, we develop a calculus for concurrent constraint programming which has the same observable behavior as the operational semantics defined in the last section. The calculus is formulated as a reduction system modulo a set of structural congruences.

The calculus describes a slightly larger language than ccp, adding a blind choice operator $(+)$ and a failure operator $\mathbf{0}$, which is an identity for $(+)$. Informally, using $(+)$ one can collect all possible execution paths of an agent. We also admit a new form of guarded branch in an ask agent, written $\surd \to \mathbf{0}$, which stands for failure upon termination. Hence, a guard g is now a constraint c or the symbol $\surd$. Informally, once an alternative in a guarded choice is selected, the branch that corresponds to taking some other alternative is marked with a $\surd$-guard, which causes the branch to be discarded upon termination.

Example 3.1 To see the essential idea for obtaining confluence, consider the agent

$$A \stackrel{def}{=} d \mapsto M \mathbin{[\!]} e \mapsto N,$$

run in a context where the store entails d. If the store does not also entail e this should rewrite to M. On the other hand, if the store entails both d and e, A should rewrite to $M + N$. The problem is that the property "the store does not imply e" is not monotonic – in fact it is anti-monotonic since the store increases monotonically during execution. Therefore, it is not possible to make a choice between the two reductions uniformly for all process schedulings. One solution to the problem is to consider each possible process scheduling individually, using an interpretation of parallel composition as interleaving. The resulting calculus is unsuitable for program analysis, however, due to the state space explosion incurred by the interleaving semantics.

In our calculus, A reduces instead to

$$M + (e \mapsto N \mathbin{[\!]} \surd \mapsto 0) \stackrel{def}{=} B.$$

In effect this defers the decision whether or not to drop the "$e \mapsto N$" branch until program termination. If further reductions determine that the store also entails e, this term could further reduce to

$$M + N + (\surd \mapsto 0 \mathbin{[\!]} \surd \mapsto 0),$$

which is observationally equivalent to $M + N$. On the other hand, if the store never entails e, we end with agent B, which produces the same observations as M. We thus get a confluent calculus that is observationally equivalent to the transition system presented in the last section.

We now make these intuitions precise by defining a reduction system over an extended concurrent constraint language, called ccp_{+0}. Terms in ccp_{+0} are produced by the grammar.

Agent	$M ::= c \mid R \mid p\overline{y} \mid M \cdot M \mid \exists_x M \mid M + M \mid \mathbf{0}$
Choice	$R ::= R \mathbin{[\!]} R \mid c \mapsto M \mid \surd \mapsto \mathbf{0}$

The definitions of renaming and free variables carry over in the obvious way.

The operators have the natural precedence rules: $\exists_x$ binds strongest, followed by $(\cdot)$, followed by $(\,\|\,)$, followed by $(+)$ which binds weakest. Guard prefixes $g \mapsto$ extend as far to the right as possible.

The ccp calculus has a rich set of structural equivalences ($\equiv$). If $M \equiv N$, then M and N are generally identified. If we want to avoid this identification, speaking only of the concrete term syntax, we will explicitly talk about pre-agents or pre-programs. Structural equivalence ($\equiv$) is the least congruence that satisfies the laws below.

1. $(+)$ is associative and commutative, with identity $\mathbf{0}$.

$$\begin{aligned}(L+M)+N &\equiv L+(M+N)\\ M+N &\equiv N+M\\ M+\mathbf{0} &\equiv M\end{aligned}$$

2. $(\cdot)$ is associative and commutative, with identity *true* and zero $\mathbf{0}$.

$$\begin{aligned}(L\cdot M)\cdot N &\equiv L\cdot(M\cdot N)\\ M\cdot N &\equiv N\cdot M\\ M\cdot \mathit{true} &\equiv M\\ M\cdot\mathbf{0} &\equiv \mathbf{0}\end{aligned}$$

3. $(\cdot)$ distributes through $(+)$.

$$M\cdot(N_1+N_2) \equiv M\cdot N_1 + M\cdot N_2$$

4. $(\,\|\,)$ is associative and commutative.

$$\begin{aligned}(L \,\|\, M)\,\|\, N &\equiv L\,\|\,(M\,\|\,N)\\ M\,\|\,N &\equiv N\,\|\,M\end{aligned}$$

5. Parallel composition of constraints equals least upper bound.

$$c\cdot c' \equiv c \sqcup c'$$

6. The following laws govern existential quantification:

$$\begin{aligned}\exists_x(M+N) &\equiv \exists_x M + \exists_x N &&\\ M\cdot\exists_x N &\equiv \exists_x(M\cdot N) && \text{if } x\notin \mathrm{fv}(M)\\ \exists_x M &\equiv M && \text{if } x\notin\mathrm{fv}(M)\\ \exists_x M &\equiv \exists_y[y/x]M && \text{if } y\notin\mathrm{fv}(M)\\ \exists_x\exists_y M &\equiv \exists_y.\exists_x M &&\end{aligned}$$

Reduction $\rightarrow$ is a binary relation between agents that is parameterized by a procedure environment D. We write $M \rightarrow_D N$ if M reduces to N in one step in the procedure environment D. We sometimes leave out the D-suffix if the environment is clear from the context.

In essence there are two reduction rules, one for communication, and one for procedure unfolding. The rule for procedure unfolding is:

$$p\overline{y} \xrightarrow{p}_D [\overline{y}/\overline{x}]M \qquad (p\overline{x} := M \;\in\; D).$$

The rule for communication comes in two variants. The first variant handles the deterministic case, where no choice operator is present:

$$c \cdot (d \mapsto M) \xrightarrow{cc}_D c \cdot M \qquad (d \leq c)$$

The second variant handles the case where the ask agent is part of a guarded choice:

$$c \cdot (d \mapsto M \parallel R) \xrightarrow{cc}_D c \cdot M + c \cdot (\surd \mapsto \mathbf{0} \parallel R) \qquad (d \leq c)$$

The standard semantics of ccp captures the idea that once a guard in one of the guarded choice branches is enabled then that branch can be chosen and the other branches can be discarded. By contrast, our rule does not discard any branches. Instead, we also keep the original ask agent as a (+)-alternative, but with the taken branch replaced by the branch ($\surd \mapsto \mathbf{0}$). Essentially this indicates that the alternative cannot lead to suspension, but that other branches in the alternative can still be taken if their guards are enabled.

Reduction can only occur in the top-level agents, it cannot occur inside the branches of a guarded choice. That is, our reduction relation, $\rightarrow$, is given by

$$\frac{M \xrightarrow{p \cup cc}_D M'}{\exists_{\overline{x}}(M \cdot N) + N' \rightarrow_D \exists_{\overline{x}}(M' \cdot N) + N'}.$$

We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\rightarrow$.

We now define the set of possible observations of a ccp-term M. Since we express non-determinism by the (+) operator, we might expect that each (+)-alternative in a reduct would contribute to the set of possible observations. However, we have to disregard those alternatives that contain a guard of the form $\surd \mapsto \mathbf{0}$ at top-level, since they represent untaken branches in a committed choice. Upon termination such alternatives are identified with failure, as is formalized below.

Definition. Let *terminal equivalence* $\approx$ be the least congruence that contains $\equiv$ and the equality

$$R \parallel \surd \mapsto \mathbf{0} \;\approx\; \mathbf{0}.$$

Definition. The *constraint part* $Con(M)$ of a term M is $\bigsqcup\{c \mid \exists N.M \equiv c \cdot N\}$.

Definition. A term M is in *normal form* if it cannot be reduced by $\rightarrow_D$.

Definition. Let P be the ccp_{+0} program D ; M. Then $P \Downarrow_{\mathsf{ccp}_{+0}} c$ if there is a normal form N and a term M' such that $M \twoheadrightarrow_D N + M'$, $N \not\approx \mathbf{0}$ and $c = Con(N)$. P *diverges*, written $P \Uparrow_{\mathsf{ccp}_{+0}}$ if there is an infinite $\rightarrow_D$-transition sequence starting with M.

The set of *observations* of a program P, $Obs(\rightarrow, P)$ is defined as in the ccp case.

$$Obs(\rightarrow, P) = \{c \mid M \Downarrow_{\mathsf{ccp}_{+0}} c\} \cup \{\mathit{false} \mid M \Uparrow_{\mathsf{ccp}_{+0}}\}.$$

Thus, the possible observations of a program P are the constraint parts of all non-zero normal form alternatives of P. In addition, we add *false* to the observations of P if there is a possibility that evaluation of P does not terminate. We often abbreviate $Obs(\rightarrow, P)$ to $Obs(P)$.

As usual, we define *observational equivalence* ($\cong$) to be the largest congruence on terms and programs such that $P \cong Q$ implies $Obs(P) = Obs(Q)$, for all programs P, Q.

An equivalent, but more constructive definition of $\cong$ for terms is based on a *program context*, C, which is a program with a hole $[\,]$ in it. Let $C[M]$ denote the term that results from filling out the hole in C. Then $M \cong N$ iff for all program contexts C such that $C[M]$ and $C[N]$ are well-formed programs,

$$Obs(C[M]) = Obs(C[N]).$$

Proposition 3.2 The following are observational equivalences in ccp_{+0}.

$$\begin{aligned} M + M &\cong M \\ M_1 + M_2 &\cong true \mapsto M_1 \,\|\, true \mapsto M_2 \\ R \,\|\, R &\cong R \\ c \cdot (d \mapsto M \,\|\, R) &\cong c \cdot R && (c \sqcup d = \mathit{false}) \\ c \cdot (d \mapsto M \,\|\, R \,\|\, \surd \mapsto \mathbf{0}) &\cong c \cdot (d \mapsto M \,\|\, R) && (d \leq c) \end{aligned}$$

Note that the second observational equivalence means that the explicit blind choice construct does not add to the expressiveness of ccp.

Example 3.3 A reduction sequence in ccp_{+0} using left-most agent scheduling from the program given in Example 2.2 is given in Figure 2, where M, M' and M'' are appropriate renamings of the definition of $merge(x, y, z)$, $merge(x', y, z')$ and $merge(x, y', z')$ respectively and R' and R'' are the remaining branches in the guarded choices in M' and M''. This reduction sequence gives the observable behavior

$$\{y = [b] \sqcup x = [a] \sqcup z = [b, a], y = [b] \sqcup x = [a] \sqcup z = [a, b]\}.$$

This is exactly the observable behaviour with the ccp operational semantics, but is obtained with a single reduction scheduling.

4 Confluence

In this section we show that $\rightarrow$ is confluent. The confluence proof has to overcome the difficulty that agents do not form a free algebra (modulo α-renaming), but are equivalence classes of pre-agents. Hence, standard techniques such as studied in [8] or [9] are not applicable.

Instead we adopt the following strategy: We define a *canonical form* $[\![M]\!]$ of a term M, together with a reduction relation on canonical forms. We show that the canonical form mapping has an inverse, and that both it and its inverse commute with equivalences and multi-step reductions. We then show that reduction on

$$
\begin{array}{ll}
 & x = [a] \cdot merge(x, y, z) \cdot y = [b] \\
\xrightarrow{p} & y = [b] \sqcup x = [a] \cdot M \\
\xrightarrow{cc} & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'} \exists_u \exists_{z'}\ (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x', y, z')) \\
 & +\exists_{y'} \exists_u \exists_{z'}\ (y = [u \mid y'] \cdot z = [u \mid z'] \cdot merge(x, y', z')) \\
 & +\surd \mapsto \mathbf{0} \ [\!]\ x = [] \mapsto z = y \ [\!]\ y = [] \mapsto z = x \\
 &) \\
\cong & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'} \exists_u \exists_{z'}\ (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x', y, z')) \\
 & +\exists_{y'} \exists_u \exists_{z'}\ (y = [u \mid y'] \cdot z = [u \mid z'] \cdot merge(x, y', z')) \\
 &) \\
\xrightarrow{p} & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'} \exists_u \exists_{z'}\ (x = [u \mid x'] \sqcup z = [u \mid z'] \cdot M') \\
 & +\exists_{y'} \exists_u \exists_{z'}\ (y = [u \mid y'] \sqcup z = [u \mid z'] \cdot M'') \\
 &) \\
\xrightarrow{cc} & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'} \exists_u \exists_{z'}\ (x = [u \mid x'] \sqcup z = [u \mid z'] \cdot (z' = y + \surd \mapsto \mathbf{0} \ [\!]\ R')) \\
 & +\exists_{y'} \exists_u \exists_{z'}\ (y = [u \mid y'] \sqcup z = [u \mid z'] \cdot (z' = x + \surd \mapsto \mathbf{0} \ [\!]\ R'')) \\
 &) \\
\cong & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'} \exists_u \exists_{z'}\ (x = [u \mid x'] \sqcup z = [u \mid z'] \sqcup z' = y) \\
 & +\exists_{y'} \exists_u \exists_{z'}\ (y = [u \mid y'] \sqcup z = [u \mid z'] \sqcup z' = x) \\
 &) \\
\equiv & y = [b] \sqcup x = [a] \sqcup z = [a, b] + y = [b] \sqcup x = [a] \sqcup z = [b, a].
\end{array}
$$

Fig. 2. Example reduction in ccp_{+0}.

canonical forms is confluent, using standard techniques. By the properties of the canonical form mapping, this gives us then confluence of the original ccp_{+0} calculus. A similar technique has been used by Niehren and Smolka in their confluence proofs for the δ and ρ calculi [13, 14].

Definition. A *canonical form* X is a multi-set of *alternatives*. Each alternative A is a quadruple (xs, c, ps, rs), where

- xs is a set of variables (the bound variables of the alternative).
- c is a constraint.
- ps is a multi-set of procedure calls $p\overline{y}$.
- rs is a multi-set of *readers*, where each reader is itself a non-empty multi-set of pairs (g, X), with g a guard and X a canonical form. We assume that the termination guard $\surd$ appears only in conjunction with the empty set (which represents $\mathbf{0}$).

Let letters X, Y, Z range over canonical forms.

The set of free variables $\mathrm{fv}(X)$ of a canonical form X is the union of the sets of free variables of its alternatives. The free variables of an alternative (xs, c, ps, rs) is the union of the free variables of its components, minus all variables that occur in xs. We assume that for each alternative (xs, c, ps, rs) in a canonical form it holds that $xs \subseteq \mathrm{fv}(\emptyset, c, ps, rs)$.

Two alternatives $A \stackrel{def}{=} (xs, c, ps, rs)$ and $B \stackrel{def}{=} (ys, d, qs, ss)$ are considered identical if $xs \cap \mathrm{fv}(B) = ys \cap \mathrm{fv}(A) = \emptyset$ and there exists a renaming ρ from xs to ys such that $B = \rho A$.

Definition. A *canonical form environment* is a set of procedure definitions $\{p\overline{x} = X\}$ that associate a procedure name p and formal arguments $\overline{x}$ with a canonical form X. We use the letter E for canonical form environments.

We now define some useful operations on canonical forms and alternatives. Let

$$A \stackrel{def}{=} (xs, c, ps, rs)$$
$$B \stackrel{def}{=} (ys, d, qs, ss)$$

be two alternatives such that $xs \cap ys = xs \cap \mathrm{fv}(B) = ys \cap \mathrm{fv}(A) = \emptyset$. Then their least upper bound is given by

$$A \sqcup B = ((xs \cup ys) \cap \mathrm{fv}(\emptyset, c \sqcup d, ps \cup qs, rs \cup ss), c \sqcup d, ps \cup qs, rs \cup ss).$$

Existential quantification $\exists_x A$ of an alternative A is defined as follows.

$$\exists_x(xs, c, ps, rs) = \begin{cases} (xs, c, ps, rs) & \text{if } x \notin \mathrm{fv}(xs, c, ps, rs) \\ (xs, \exists_x c, ps, rs) & \text{if } x \in \mathrm{fv}(c) \wedge x \notin \mathrm{fv}(xs, \mathit{true}, ps, rs) \\ (xs \cup \{x\}, c, ps, rs) & \text{otherwise.} \end{cases}$$

Another useful operation is the merge $\uplus$ of two alternatives with a single reader each into an alternative where both readers are combined.

$$(xs, c, ps, \{r_1\}) \uplus (xs, c, ps, \{r_2\}) = (xs, c, ps, \{\bigcup(r_1 \cup r_2)\}).$$

$$\begin{aligned}
[\![c]\!] &= \{(\emptyset, c, \emptyset, \emptyset)\} \\
[\![p\overline{y}]\!] &= \{(\emptyset, \mathit{true}, \{p\overline{y}\}, \emptyset)\} \\
[\![\exists_x M]\!] &= \{\exists_x A \mid A \in [\![M]\!]\} \\
[\![M \cdot N]\!] &= \{A \sqcup B \mid A \in [\![M]\!], B \in [\![N]\!]\} \\
[\![M + N]\!] &= [\![M]\!] \cup [\![N]\!] \\
[\![\mathbf{0}]\!] &= \emptyset \\
[\![g \mapsto M]\!] &= \{(\emptyset, \mathit{true}, \emptyset, \{\{(g, [\![M]\!])\}\})\} \\
[\![R \parallel S]\!] &= \{A \uplus B \mid A \in [\![R]\!], B \in [\![S]\!]\}
\end{aligned}$$

$$\begin{aligned}
[\![\emptyset]\!]^{-1} &= \mathbf{0} \\
[\![\{A_1, ..., A_n\}]\!]^{-1} &= [\![A_1]\!]^{-1} + ... + [\![A_n]\!]^{-1} \qquad (n \geq 1) \\
[\![(\overline{x}, c, \{p_1\overline{y_1}, ..., p_j\overline{y_j}\}, \{r_1, ..., r_k\})]\!]^{-1} &= \exists_{\overline{x}}(c \cdot p_1\overline{y_1} \cdot \ldots \cdot p_j\overline{y_j} \cdot [\![r_1]\!]^{-1} \cdot \ldots \cdot [\![r_k]\!]^{-1}) \\
[\![\{(g_1, X_1), \ldots, (g_m, X_m)\}]\!]^{-1} &= g_1 \mapsto [\![X_1]\!]^{-1} \parallel \ldots \parallel g_m \mapsto [\![X_m]\!]^{-1}
\end{aligned}$$

Fig. 3. Mapping a term to its canonical form and back.

Figure 3 presents a mapping $[\![\cdot]\!]$ that maps a pre-term to its canonical form, together with its right inverse, $[\![\cdot]\!]^{-1}$.

Lemma 4.1 For all pre-terms M, N, we have $M \equiv N$ iff $[\![M]\!] = [\![N]\!]$.

We now define a notion of reduction $\Rightarrow$ on canonical forms that simulates reduction $\rightarrow$ on ccp_{+0} terms. Analogous to $\rightarrow$, $\Rightarrow$ is parameterized by a normal form environment. There are three different ways a canonical form X can reduce.

1. If $(p\overline{x} = Y) \in E$ and X is $X' \cup \{(xs, c, \{p\overline{y}\} \cup ps, rs)\}$ then

$$X \Rightarrow_E X' \cup \{(xs, c, ps, rs) \sqcup A \mid A \in [\overline{y}/\overline{x}]Y\}.$$

2. If $d \leq c$ and X is $X' \cup \{(xs, c, ps, \{\{(d, Y)\}\} \cup rs)\}$ then

$$X \Rightarrow_E X' \cup \{(xs, c, ps, rs) \sqcup A \mid A \in Y\}.$$

3. If $r \neq \emptyset$, $d \leq c$ and X is $X' \cup \{(xs, c, ps, \{\{(d, Y)\} \cup r\} \cup rs)\}$ then

$$X \Rightarrow_E X' \cup \{(xs, c, ps, rs) \sqcup A \mid A \in Y\} \cup \{(xs, c, ps, \{\{(\surd, \mathbf{0})\} \cup r\} \cup rs\})\}.$$

We now show that multi-step $\Rightarrow$ reduction can simulate $\rightarrow$.

Lemma 4.2 For all terms M, N, procedure environments D, if $M \rightarrow_D N$, then $[\![M]\!] \Rightarrow\!\!\!\!\Rightarrow_{[\![D]\!]} [\![N]\!]$.

The reverse of Lemma 4.2 also holds.

Lemma 4.3 For all canonical forms X, Y, canonical form environments E, if $X \Rightarrow_E Y$, then $[\![X]\!]^{-1} \rightarrow_{[\![E]\!]^{-1}} [\![Y]\!]^{-1}$.

We now establish that reduction $\Rightarrow$ is confluent.

Definition. Let $\stackrel{p}{\Longrightarrow}$ be the reduction relation generated by the first rule (the unfolding rule) in the definition of $\Rightarrow$. Let $\stackrel{cc}{\Longrightarrow}$ be the reduction relation generated by the second and third rule (the communication rules) in the definition of $\Rightarrow$.

Lemma 4.4 $\stackrel{p}{\Longrightarrow}$ is Church-Rosser:
If $X \stackrel{p}{\Longrightarrow\!\!\!\!\Rightarrow}_E X_1$ and $X \stackrel{p}{\Longrightarrow\!\!\!\!\Rightarrow}_E X_2$ then there is a canonical form X_3 s.t. $X_1 \stackrel{p}{\Longrightarrow\!\!\!\!\Rightarrow}_E X_3$ and $X_2 \stackrel{p}{\Longrightarrow\!\!\!\!\Rightarrow}_E X_3$.

Lemma 4.5 $\stackrel{cc}{\Longrightarrow}$ is Church-Rosser.

Lemma 4.6 $\Rightarrow$ is Church-Rosser.

Proof: By Lemma 4.4 and Lemma 4.5, $\stackrel{p}{\Longrightarrow}$ and $\stackrel{cc}{\Longrightarrow}$ are both Church-Rosser. An analysis of reduction sequences shows that $\stackrel{p}{\Longrightarrow}$ and $\stackrel{cc}{\Longrightarrow}$ commute. By the Lemma of Hindley and Rosen [1, Prop. 3.3.5], it follows that $\Rightarrow = \stackrel{p}{\Longrightarrow} \cup \stackrel{cc}{\Longrightarrow}$ is Church-Rosser. $\square$

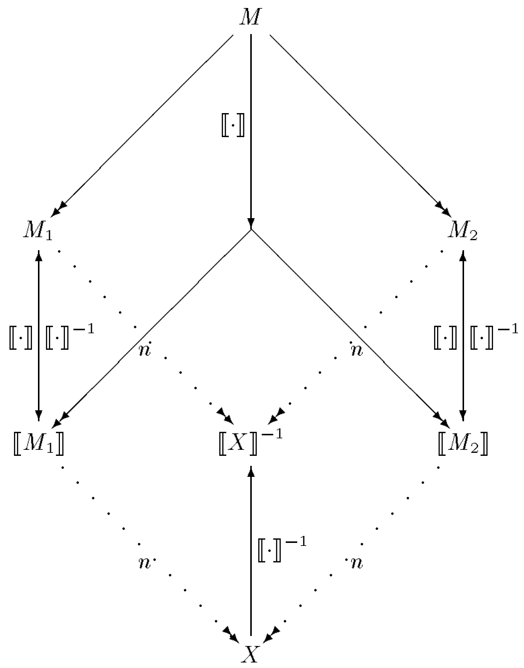

Fig. 4. Strategy of the CR proof

We are finally in a position to show confluence for the original notion of reduction $\rightarrow$ on ccp_{+0} terms.

Theorem 4.7 $\rightarrow$ is Church-Rosser. For all terms M, M_1, M_2, environments D, if $M \twoheadrightarrow_D M_1$ and $M \twoheadrightarrow_D M_2$ then there is a term M_3 s.t. $M_1 \twoheadrightarrow_D M_3$ and $M_2 \twoheadrightarrow_D M_3$.

Proof: The proof strategy is depicted in Figure 4. Assume that $M \twoheadrightarrow_D M_1$ and $M \twoheadrightarrow_D M_2$. By an induction on the length of the two reduction sequences from M to M_1 and M_2, using Lemma 4.1 and Lemma 4.2 at each step, we have that $\llbracket M \rrbracket \Rightarrow\!\!\!\!\Rightarrow_{\llbracket D \rrbracket} \llbracket M_1 \rrbracket$ and $\llbracket M \rrbracket \Rightarrow\!\!\!\!\Rightarrow_{\llbracket D \rrbracket} \llbracket M_2 \rrbracket$. Since by Lemma 4.6 $\Rightarrow$ is confluent, this implies the existence of a canonical form X such that $\llbracket M_1 \rrbracket \Rightarrow\!\!\!\!\Rightarrow_{\llbracket D \rrbracket} X$ and $\llbracket M_2 \rrbracket \Rightarrow\!\!\!\!\Rightarrow_{\llbracket D \rrbracket} X$. As $\llbracket \cdot \rrbracket^{-1}$ is an inverse of $\llbracket \cdot \rrbracket$, $\llbracket \llbracket M_i \rrbracket \rrbracket^{-1} \equiv M_i$ for $i = 1, 2$. Then by induction on the length of the two reduction sequences from $\llbracket M_1 \rrbracket$ and $\llbracket M_2 \rrbracket$ to X, using Lemma 4.3 at each step, we have that $M_i = \llbracket \llbracket M_i \rrbracket \rrbracket^{-1} \twoheadrightarrow \llbracket X \rrbracket^{-1}$, $(i = 1, 2)$. This implies the proposition with $M_3 = \llbracket X \rrbracket^{-1}$. $\square$

5 Relationship to ccp

In this section we show that the observational behaviour of our calculus is identical to the observational behaviour of ccp in its standard transition system semantics. To do this we extend $\llbracket \cdot \rrbracket$ so that it maps a ccp configuration to a subset of the canonical forms given in the previous section, together with a reduction relation $\stackrel{ccp}{\Longrightarrow}$ on this canonical form and a notion of observables. We show that for a given program $\stackrel{ccp}{\longrightarrow}$, $\stackrel{ccp}{\Longrightarrow}$, $\Rightarrow$ and $\rightarrow$ all give rise to the same observations.

In order to extend $[\![\cdot]\!]$, we first give a mapping $pa()$ from ccp agents in a configuration to a ccp_{+0} pre-agent. This is needed because ccp agents in a configuration may have hidden stores which are not allowed in pre-agents.

$$\begin{array}{ll} pa(c) & = c \\ pa(p\overline{y}) & = p\overline{y} \\ pa(\exists_x^d M) & = \exists_x(d \cdot pa(M)) \\ pa(M \cdot N) & = pa(M) \cdot pa(N) \\ pa(g \mapsto M) & = g \mapsto M. \end{array}$$

Note that terms in the range of pa never contain $\mathbf{0}$, $+$ or $\surd$. The *canonical form* of a ccp configuration $\langle A, c\rangle$ is given by

$$[\![\langle A, c\rangle]\!] = [\![pa(A) \cdot c]\!].$$

As ccp agents and programs do not contain blind choice, the canonical form of a ccp configuration will always consist of a single alternative. Because there is no need to distribute blind choice over the parallel operator, there is a bijection between the readers and the procedure calls in the ccp configuration and the canonical form. We will make use of this correspondence in the proofs below.

We now define a notion of reduction $\stackrel{ccp}{\Longrightarrow}$ on the canonical form of a ccp agent that simulates reduction $\stackrel{ccp}{\longrightarrow}$ on ccp configurations. Like $\stackrel{ccp}{\longrightarrow}$, $\stackrel{ccp}{\Longrightarrow}$ is parameterized by an environment E of definitions, i.e., associations between ccp procedure names with formal arguments and canonical forms. There are two different ways a ccp canonical form X can reduce:

1. If $(p\overline{x} = \{A\}) \in E$ then

$$\{(xs, c, \{p\overline{y}\} \cup ps, rs)\} \stackrel{ccp}{\Longrightarrow}_E \{(xs, c, ps, rs) \sqcup A\}.$$

2. If $d \leq c$ then

$$\{(xs, c, ps, \{\{(d, \{A\})\}\} \cup rs)\} \stackrel{ccp}{\Longrightarrow}_E \{(xs, c, ps, rs) \sqcup A\}.$$

Definition. A canonical form is in *normal form* if it cannot be reduced. $Con(A)$ is the constraint component of A. We write $\stackrel{ccp}{\Longrightarrow\!\!\!\!\!\!\rightarrow}$ for the reflexive and transitive closure of $\stackrel{ccp}{\Longrightarrow}$.

Analogous to the cases for $\rightarrow$ reductions and ($\stackrel{ccp}{\longrightarrow}$) transitions, we now define two notions of observables for canonical form reductions.

Definition. Let the notion of reduction $\hookrightarrow$ be one of $\Rightarrow$, $\stackrel{ccp}{\Longrightarrow}$. Let P be the ccp program D ; M. Then the set of possible observations of P wrt $\hookrightarrow$ is given by

$$Obs(\hookrightarrow, P) = \bigcup\{Obs([\![A]\!]^{-1}) \mid P \hookrightarrow\!\!\!\!\rightarrow \{A\} \cup X \ \wedge\ \{A\} \text{ is in } \hookrightarrow\text{-normal form}\}.$$

The following two lemmas are shown by an analysis of $\xrightarrow{ccp}$ transitions and $\xRightarrow{ccp}$ reductions.

Lemma 5.1 If $S \xrightarrow{ccp} S'$ in the transition system T_D, then either $[\![S]\!] = [\![S']\!]$ or $[\![S]\!] \xRightarrow{ccp}_{[\![D]\!]} [\![S']\!]$.

Lemma 5.2 Let S be a ccp configuration and D be a set of ccp definitions. If $[\![S]\!] \xRightarrow{ccp}_{[\![D]\!]} X$ then there is a configuration S' such that $X = [\![S']\!]$ and $S \xrightarrow{ccp}\!\!\!\!\rightarrow S'$ in the transition system T_D.

Thus:

Lemma 5.3 For any ccp program P, $Obs(\xrightarrow{ccp}, P) = Obs(\xRightarrow{ccp}, P)$.

We also have that:

Lemma 5.4 For any ccp program P, $Obs(\xRightarrow{ccp}, P) = Obs(\Rightarrow, P)$.

Lemma 5.5 For any program P, $Obs(\Rightarrow, P) = Obs(P)$.

The main result of this section follows from Lemma 5.3, Lemma 5.4 and Lemma 5.5 – the confluent calculus is observationally equivalent to the operational semantics of ccp.

Theorem 5.6 For any ccp program P, $Obs(P) = Obs(\xrightarrow{ccp}, P)$.

6 Application to Program Analysis

One application of our confluent semantics is to the static analysis of ccp programs. Codish *et al* [3, 2] propose a generic approach to the analysis of concurrent logic and constraint programs. They introduce a confluent semantics which approximates the standard (non-confluent) semantics of the concurrent constraint logic languages and use this as a basis for program analysis. Correctness of their analysis holds because the confluent semantics approximates the standard semantics in the sense that any successful reduction sequence in the usual semantics is also a valid reduction sequence in the confluent semantics, and suspension in the usual semantics implies suspension in the confluent semantics. The reason for requiring confluence is so that an analysis based on this semantics need only be proven correct for a single scheduling rule. This provides for accuracy as the analysis can choose a scheduling which gives the most precise answer and also provides for efficiency as there is no need to examine the potentially exponential or even infinite number of different but "isomorphic" reduction sequences corresponding to other schedulings. Zaffanella et al [17] and Falaschi et al [6] have given a modification of this idea for the slightly different context of ccp. They formalize the analysis as a transformation from a program written in full ccp to a an approximating program written in a subset of ccp for which the usual operational semantics is confluent.

Our calculus provides an alternative semantic basis for program analysis. Because the calculus is Church-Rosser it has all of the advantages of the approximate confluent semantics or program transformation. Yet it is inherently more precise because programs have exactly the same observable behaviour as in the usual operational semantics and the calculus does not introduce extra reductions. For example, consider the ccp agent

$$p(x) \cdot choose(x, y, z) \cdot c(z)$$

with the following ccp definitions.

$$\begin{aligned}
p(x) &:= x = a \\
choose(x, y, z) &:= x = a \mapsto z = x \parallel y = a \mapsto z = y \\
c(z) &:= z = a \mapsto true
\end{aligned}$$

No analysis based on the approximate confluent semantics or transformed program approach can ever prove that this agent is suspension free as the approximate operational semantics and program transformation introduce a reduction sequence which leads to suspension. However, an analysis based on our calculus can show that this agent does not lead to suspension.

7 Conclusion

We have given a calculus for a class of languages, ccp_{+0}, which generalize concurrent constraint programs (ccp). However, unlike the usual operational semantics for ccp, the calculus is confluent in the sense that different process schedulings give rise to exactly the same set of possible outcomes. This disproves the folklore that it is impossible to give a confluent semantics for languages with non-deterministic guarded choice.

The calculus has application to static analysis of ccp programs. As the calculus is confluent, it provides a good basis on which develop analyses. Confluence means that not all process schedulings need to be considered in an analysis, allowing for efficiency, and that an analysis can choose a process scheduling which gives better information, allowing for accuracy.

Acknowledgements

We thank the referees for their detailed comments.

References

1. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

2. M. Codish, M. Falaschi, and K. Marriott. Suspension analyses for concurrent logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
3. M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient analysis of concurrent constraint logic programs. In *Proc. 20th International Colloquium on Automata, Languages, and Programming*, pages 633–644. Springer Verlag, 1993. LNCS 700.
4. C. Codognet, P. Codognet, and M. Corsini. Abstract interpretation for concurrent logic languages. In *Proc. North American Conf. on Logic Programming*, pages 215–232, 1990.
5. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *Proc. 8th IEEE Symposium on Logic In Computer Science*, pages 210–221, 1993.
6. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and concurrent constraint programming. In *to appear in Proc. AMAST*, 1995.
7. Leon Henkin, J.Donald Monk, and Alfred Tarski. *Cylindric Algebras*, volume 64 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1971.
8. Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
9. Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematisch Centrum, Kruislaan 413, 1098 SJ Amsterdam, 1980. Mathematical Centre Tracts n. 127.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.
11. M. Maher. Logic semantics for a class of committed-choice programs. In *Proc. Fouth Int. Conf. on Logic Programming*, pages 858–876, 1987.
12. U. Montanari, F. Rossi, and V. Saraswat. CC programs with both in- and non-determinism: A concurrent semantics. In *Second Int. Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, pages 151–161. Springer Verlag, 1994. LNCS 874.
13. Joachim Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen*. PhD thesis, Universität des Saarlandes, 1994.
14. Joachim Niehren and Gert Smolka. A confluent relational calculus for higher-order programming with constraints. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 89–104, München, Germany, 7–9 September 1994. Springer-Verlag.
15. Vijay Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 333–352. ACM Press, January 1991.
16. Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, California, January 1990.
17. E. Zaffanella, G. Levi, and R. Giacobazzi. Abstracting synchronization in concurrent constraint programming. In *Proc. 5th Int'l Symposium on Programming Language Implementation and Logic Programming*. Springer Verlag, 1994. LNCS 844.

This article was processed using the LaTeX macro package with LLNCS style