Unfold/Fold Transformations of CCP Programs

Sandro Etalle[1], Maurizio Gabbrielli[2], Maria Chiara Meo[3]

[1] Universiteit Maastricht, P.O. Box 616, 6200MD Maastricht, The Netherlands.
etalle@cs.unimaas.nl
[2] Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.
gabbri@di.unipi.it.
[3] Diaprtimento di Matematica Pura e Applicata, Università di L'Aquila, Loc. Coppito, 67010 L'Aquila, Italy. meo@univaq.it.[†]

Abstract. We introduce a transformation system for concurrent constraint programming (CCP). We define suitable applicability conditions for the transformations which guarantee that the input/output ccp semantics is preserved also when distinguishing deadlocked computations from successful ones.
The systems allows to optimize CCP programs while preserving their intended meaning. Furthermore, since it preserves the deadlock behaviour of programs, it can be used for proving deadlock freeness of a class of queries in a given program.

Keywords: Transformation, Concurrent Constraint Programming, Deadlock.

1 Introduction

Optimization techniques, in the case of logic-based languages, fall into two main categories: on one hand, there exist methods for compile-time and low-level optimizations such as the ones presented for constraint logic programs in [10], which are usually based on program analysis methodologies (e.g. abstract interpretation). On the other hand, we find source to source transformation techniques such as *partial evaluation* (see [15]) (which in the field of logic programming is mostly referred to as *partial deduction* and is due to Komorowski [11]), and more general techniques based on the *unfold* and *fold* or on the *replacement* operation.

Unfold/fold transformation techniques were first introduced for functional programs in [2], and then adapted to logic programming (LP) both for program synthesis [3, 9], and for program specialization and optimization [11]. Tamaki and Sato in [22] proposed a general framework for the unfold/fold transformation of logic programs, which has remained in the years the main historical reference of the field, and has recently been extended to constraint logic programming (CLP) in [1, 5, 13] (for an overview of the subject, see the survey by Pettorossi and Proietti [16]). As shown by a number of applications, these techniques provide powerful methodology for the development and optimization of large programs, and can be regarded as the *basic* transformations techniques, which might be further adapted to be used for partial evaluation.

[†] The work of the third author was partially supported by the MURST 40% project: Tecniche speciali per la verifica, l'analisi, la sintesi e la trasformazione di programmi".

Despite a large literature in the field of sequential languages, unfold/fold transformation sequences have hardly been applied to concurrent logic languages. Notable exceptions are the papers of Ueda and Fukurawa [23], Sahlin [17], and of de Francesco and Santone [8] (their relations with this paper are discussed in Section 5). This situation is partially due to the fact that the non-determinism and the synchronization mechanisms present in concurrent languages substantially complicate their semantics, thus complicating also the definition of *correct* transformation systems. Nevertheless, as argued below, transformation techniques can be be more useful for concurrent languages than they already are for sequential ones.

In this paper we introduce a transformation system for concurrent constraint programming (CCP) [18, 19, 20]. This paradigm derives from replacing the *store-as-valuation* concept of von Neumann computing by the *store-as-constraint* model: Its computational model is based on a global *store*, which consists of the conjunction of all the constraints established until that moment and expresses some partial information on the values of the variables involved in the computation. Concurrent processes synchronize and communicate asynchronously via the store by using elementary actions (ask and tell) which can be expressed in a logical form (essentially implication and conjunction [4]). On one hand, CCP enjoys a clean logical semantics, avoiding many of the complications arising in the concurrent imperative setting; as argued in the position paper [6] this aspect is of great help in the development of effective transformation (and partial evaluation) tools. On the other hand, CCP benefits of a number of existing implementations, an example being Oz [21]; thus, in contrast to other models for concurrency such as the π-calculus, in this framework transformation techniques can be readily applied to practical problems.

The transformation system we are going to introduce is originally inspired by the system of Tamaki and Sato [22], on which it improves in three main ways: firstly, by taking full advantage of the flexibility and expressivity of CCP, it introduces a number of new important transformation operations, allowing optimizations that would not be possible in the LP or CLP context; secondly, our system we managed to eliminate the limitation that in a folding operation the *folding clause* has to be nonrecursive, a limitation which is present in virtually all other unfold/fold transformation systems, this improvement possibly leads to the use of new more sophisticated transformation strategies; finally, the applicability conditions we propose for the folding operation are now independent from the *transformation history*, making the operation much easier to understand and, possibly, to be implemented.

We will show show with a practical example how our transformation system for CCP can be even more useful than its predecessors for sequential logic languages. Indeed, in addition to the usual benefits, in this context the transformations can also yield to the elimination of communication channels and of synchronization points, to the transformation of non-deterministic computations into deterministic ones, and to the crucial saving of computational *space*. It is also worth mentioning that the declarative nature of CCP allows us to define reasonably simple applicability conditions which ensure the correctness of our system.

Our results show that the original and the transformed program have the same input/output behaviour both for successful and for deadlocked derivations. As a corollary, we obtain that the original program is deadlock free iff the transformed one is, and this allows to employ the transformation as an effective tool for proving

deadlock-freeness: if, after the transformation, we can prove or see that the process we are considering never deadlocks (in some cases the transformation simplifies the program's behaviour so that this can be immediately checked), then we are also sure that does not deadlock before the transformation either.

2 Preliminaries

The basic idea underlying CCP is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can *add* a constraint c to the store by performing the basic action tell(c). Dually, agents can also *check* whether a constraint c is entailed by the store by using an ask(c) action. This allows the synchronization of different agents.

Concurrent constraint languages are defined parametrically wrt to the notion of *constraint system*, which is usually formalized in an abstract way and is provided along with the guidelines of Scott's treatment of information systems (see [19]). Here, we consider a more concrete notion of constraint which is based on first-order logic and which coincides with the one used for constraint logic programming. This will allow us to define the transformation operations in a more comprehensible way, while retaining a sufficient expressive power. Thus a *constraint* c is a first-order formula built by using predefined predicates (called primitive constraints) over a computational domain $\mathcal{D}$. Formally, $\mathcal{D}$ is a *structure* which determines the interpretation of the constraints.

In the sequel, terms will be indicated with $t, s, \ldots$, variables with $X, Y, Z, \ldots$, further, as a notational convention, $\tilde{t}$ and $\tilde{X}$ denote a tuple of terms and a tuple of distinct variables, respectively. $\exists_{-\tilde{X}}\, c$ stands for the existential closure of c *except* for the variables in $\tilde{X}$ which remain unquantified. The formula $\mathcal{D} \models \exists_{-\tilde{X}} c$ states that $\exists_{-\tilde{X}} c$ is valid in the interpretation provided by $\mathcal{D}$, i.e. that it is true for every binding in the free variables of $\exists_{-\tilde{X}} c$. The empty conjunction of primitive constraints will be identified with true. We also denote $Var(e)$ the set of variables occurring in the expression e.

The notation and the semantics of programs and agents is virtually the same one of [19]. In particular, the $\|$ operator allows one to express parallel composition of two agents and it is usually described in terms of interleaving, while nondeterminism arises by introducing a (global) choice operator $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$: the agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ nondeterministically selects one $\mathsf{ask}(c_i)$ which is enabled in the current store, and then behaves like A_i. Thus, the syntax of CCP *declarations* and *agents* is given by the following grammar:

$$
\begin{array}{lll}
\textit{Declarations} & \mathsf{D} ::= & \epsilon \mid \mathsf{p}(\tilde{t}) \leftarrow \mathsf{A} \mid \mathsf{D}, \mathsf{D} \\
\textit{Agents} & \mathsf{A} ::= & \mathsf{stop} \mid \mathsf{tell}(\mathsf{c}) \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow \mathsf{A}_i \mid \mathsf{A} \parallel \mathsf{A} \mid \mathsf{p}(\tilde{t}) \\
\textit{Processes} & \mathsf{Proc} ::= & \mathsf{D}.\mathsf{A}
\end{array}
$$

where c and c_i's are constraints. Note that, differently from [19], here we allow terms as arguments to predicate symbols. Due to the presence of an explicit choice operator, as usual we assume (without loss of generality) that each predicate symbol

is defined by exactly one declaration. In the following, following the usual practice, we call program a set of declarations.

An important aspect for which we slightly depart from the usual formalization of CCP regards the notion of *locality*. In [19] locality is obtained by using the operator $\exists$, and the behaviour of the agent $\exists_X A$ is defined like the one of A, with the variable X considered as *local* to it. Here we do not use such an explicit operator: analogously to the standard CLP setting, locality is introduced implicitly by assuming that if a process is defined by $p(\tilde{X}) \leftarrow A$ and a variable Y occurs in A but not in $\tilde{X}$, then Y has to be considered local to A.

The operational model of CCP is described by a transition system $T = (Conf, \rightarrow)$ where configurations (in) Conf are pairs consisting of a process and a constraint (representing the common *store*), while the transition relation $\rightarrow \subseteq Conf \times Conf$ is described by the (least relation satisfying the) rules **R1-R4** of Table 1 which should be self-explaining. Here and in the following we assume given a set D of declarations and we denote by $defn_D(p)$ the set of variants[5] of the (unique) declaration in D for the predicate symbol p. Due to the presence of terms as arguments to predicates symbols, differently from the standard setting in rule **R4** parameter passing is performed by a tell action. We assume also the presence of a renaming mechanism that takes care of using fresh variables each time a declaration is considered[6].

We denote by $\rightarrow^*$ the reflexive and transitive closure of the relation $\rightarrow$ defined by the transition system, and we denote by Stop any agent which contains only stop and $\|$ constructs. A finite derivation (or computation) is called *successful* if it is of the form $\langle D.A, c\rangle \rightarrow^* \langle D.Stop, d\rangle \not\rightarrow$ while it is called *deadlocked* if it is of the form $\langle D.A, c\rangle \rightarrow^* \langle D.B, d\rangle \not\rightarrow$ with B different from Stop (i.e., B contains at least one suspended agent). As it results form the transition system above, we consider here the so called "eventual tell" CCP, i.e. when adding constraints to the store (via tell operations) there is no consistency check.

R1 $\langle D.tell(c), d\rangle \rightarrow \langle D.stop, c \wedge d\rangle$

R2 $\langle D. \sum_{i=1}^{n} ask(c_i) \rightarrow A_i, d\rangle \rightarrow \langle D.A_j, d\rangle$ if $j \in [1, n]$ *and* $\mathcal{D} \models d \rightarrow c_j$

R3 $$\frac{\langle D.A, c\rangle \rightarrow \langle D.A', c'\rangle}{\begin{array}{l}\langle D.(A \parallel B), c\rangle \rightarrow \langle D.(A' \parallel B), c'\rangle \\ \langle D.(B \parallel A), c\rangle \rightarrow \langle D.(B \parallel A'), c'\rangle\end{array}}$$

R4 $\langle D.p(\tilde{t}), c\rangle \rightarrow \langle D.A \parallel tell(\tilde{t} = \tilde{s}), c\rangle$ if $p(\tilde{s}) \leftarrow A \in defn_D(p)$

Table 1. The (standard) transition system.

Using the transition system in Table 1 we define the notion of observables as follows. Here and in the sequel we say that a constraint c is *satisfiable* iff $\mathcal{D} \models \exists\, c$.

[5] A variant of a declaration d is obtained by replacing the tuple $\tilde{X}$ of all the variables appearing in d for another tuple $\tilde{Y}$.

[6] For the sake of simplicity we do not describe this renaming mechanism in the transition system. The interested reader can find in [19, 20] various formal approaches to this problem.

Definition 1 (Observables). Let D.A be a CCP process. We define

$$\mathcal{O}(\mathsf{D.A}) = \{\langle c, \exists_{-\mathsf{Var(A,c)}} d, ss\rangle \mid c \text{ and } d \text{ are satisfiable, and there exists a derivation } \langle \mathsf{D.A}, c\rangle \rightarrow^* \langle \mathsf{D.Stop}, d\rangle\}$$
$$\cup$$
$$\{\langle c, \exists_{-\mathsf{Var(A,c)}} d, dd\rangle \mid c \text{ and } d \text{ are satisfiable, and there exists a derivation } \langle \mathsf{D.A}, c\rangle \rightarrow^* \langle \mathsf{D.B}, d\rangle \not\rightarrow, \mathsf{B} \neq \mathsf{Stop}\} \ \square$$

Thus what we observe are the results of finite computations (if consistent), abstracting from the values for the local variables in the results, and distinguishing the successful computations from the deadlocked ones (by using the termination modes ss and dd, respectively). This provides the intended semantics to be preserved by the transformation system: we will call *correct* a transformation which maps a program into another one having the same observables; given the above definition, this will allow us to compare with each other the "deadlocks" and the "successes" of the original and the transformed programs.

3 The Transformation

In order to illustrate the application of our methodology we'll adopt a working example. We consider an auction problem in which two bidders participate: bidder_a and bidder_b; each bidder takes as input the list of the bids of the other one and produces as output the list of his own bids. When one of the two bidders wants to quit the auction, it produces in its own output stream the token quit. This protocol is implemented by the following program AUCTION.

```
auction(LeftBids,RightBids) ← bidder_a([0|RightBids],LeftBids) || bidder_b(LeftBids,RightBids)

bidder_a(HisList, MyList) ←
      ask(∃HisBid,HisList' HisList = [HisBid|HisList'] ∧ HisBid = quit) → stop
   + ask(∃HisBid,HisList' HisList = [HisBid|HisList'] ∧ HisBid ≠ quit) →
        tell(HisList = [HisBid|HisList']) ||
        make_new_bid_a(HisBid,MyBid) ||
           ask(MyBid = quit) → tell(MyList = [MyBid|MyList']) || broadcast("a quits")
        + ask(MyBid ≠ quit) → tell(MyList = [MyBid|MyList']) ||
              tell(MyBid ≠ quit) ||
              bidder_a(HisList',MyList')
```

plus an analogous definition for bidder_b .

Here, the agent make_new_bid_a(HisBid,MyBid) is in charge of producing a new offer in presence of the competitor's offer HisBid; the agent will produce MyBid = quit if it evaluates that HisBid is to high to be topped, and decides to leave the auction. Notice that in order to avoid deadlock, auction initializes the auction by inserting a fictitious zero bid in the input of bidder a[8].

[8] In the above program the agent tell(HisList = [HisBid|HisList']) is needed to bind the local variables (HisBid, HisList') to the global one (HisList): In fact, as resulting from the operational semantics, such a binding is not performed by the ask agent. On the

3.1 Introduction of a new definition

The introduction of a new definition is virtually always the first step of a transformation sequence. Since the new definition is going to be the main target of the transformation operation, this step will actually determine the very direction of the subsequent transformation, and thus the degree of its effectiveness.

Determining which definitions should be introduced is a potentially very difficult task which falls into the area of *strategies*. To give a simple example, if we wanted to apply *partial evaluation* to our program w.r.t. a given agent A (i.e. if we wanted to specialize our program so that it would execute the partially instantiated agent A in a more efficient way), then a good starting point would most likely be the introduction of the definition $p(\tilde{X}) \leftarrow A$, where $\tilde{X}$ is an appropriate tuple of variables and p is a new predicate symbol. Now, a different strategy would probably determine the introduction of a different new definition. For a survey of the other possibilities we refer to [16].

In this paper we are not going to be concerned with the strategies, but only with the basic transformation operations and their correctness: we aim at defining a transformation system which is general enough so to be applied in combination with different strategies. In order to simplify the terminology and the technicalities, we assume that these new declarations are added once for all to the original program before starting the transformation itself. Note that this is clearly not restrictive. As a notational convention we call D_0 the program obtained after the introduction of new definitions. In the case of program AUCTION, we assume that the following new declarations are added to the original program.

auction_left(LastBid) $\leftarrow$ tell(LastBid $\neq$ quit) || bidder_a([LastBid|Bs],As) || bidder_b(As,Bs).
auction_right(LastBid) $\leftarrow$ tell(LastBid $\neq$ quit) || bidder_a(Bs,As) || bidder_b([LastBid|As],Bs).

The agent auction_left(LastBid) engages an auction starting from the bid LastBid (which cannot be quit) and expecting the bidder "a" to be the next one in the licit. The agent auction_left(LastBid) is symmetric.

3.2 Unfolding

The first transformation we consider is the *unfolding*. This operation consists essentially in the replacement of a procedure call by its definition. The syntax of CCP agents allows us to define it in a very simple way by using the notion of context. A *context*, denoted by C[ ], is simply an agent with a "hole". C[A] denotes the agent obtained by replacing the hole in C[ ] for the agent A, in the obvious way.

Definition 2 (Unfolding). Consider a set of declarations D containing

$d: H \leftarrow C[p(\tilde{t})]$

$u: p(\tilde{s}) \leftarrow B$

Then *unfolding* $p(\tilde{t})$ in d consists simply in replacing d by

$d': H \leftarrow C[B \parallel tell(\tilde{s} = \tilde{t})]$

---

contrary the agent tell(MyBid $\neq$ quit) is redundant: We have introduced it in order to simplify the following transformations. Actually this introduction of redundant tell's is a transformation operation which is omitted here for space reasons.

in D. Here d is the *unfolded* definition and u is the unfolding one; d and u are assumed to be renamed so that they do not share variables. □

After an unfolding we often need to evaluate some of the newly introduced tell's in order to "clean up" the resulting declarations. To this aim we introduce the following operation. Here we assume that the reader is acquainted with the notion of *substitution* and of (relevant) *most general unifier* (evt. see [12]). We denote by $e\sigma$ the application of a substitution σ to an expression e.

Definition 3 (Tell evaluation). A declaration

$$d : H \leftarrow C[\mathsf{tell}(\tilde{s} = \tilde{t}) \parallel B]$$

is transformed by tell evaluation to

$$d' : H \leftarrow C[B\sigma]$$

where σ is a relevant most general unifier of s and t, and the variables in the domain[9] of σ do not occur neither in C[ ] nor in H. □

These applicability conditions can in practice be weakened by appropriately renaming some local variables. In fact, if all the occurrences of a local variable in C[ ] are in choice branches different from the one the "hole" lies in, then we can safely rename apart each one of these occurrences.

In our AUCTION example, we start working on the definition of auction_right, and we unfold the agent bidder_b([LastBid|As], Bs) and then we perform the subsequent tell evaluations. The result of these operations is the following program.

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
      ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList'] ∧ HisBid = quit) → stop
   + ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList'] ∧ HisBid ≠ quit) →
         tell([LastBid|As] = [HisBid|HisList']) ||
         make_new_bid_b(HisBid,MyBid) ||
            ask(MyBid = quit) → tell(Bs = [MyBid|Bs']) || broadcast("b quits")
         + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
               tell(MyBid ≠ quit) ||
               bidder_b(HisList',Bs')
```

3.3 Guard Simplification

A new important operation is the one which allows us to modify the ask guards occurring in a program. Consider an agent of the form $C[\mathsf{ask}(c) \rightarrow A + \mathsf{ask}(d) \rightarrow B]$ and a given set of declarations. Let us call *weakest produced constraint* of C[ ] the conjunction of all the constraints appearing in ask and tell actions which certainly have to be evaluated before [ ] is reached (in the context C[ ]). Now, if a is the context constraint of C[ ] and $\mathcal{D} \models a \rightarrow c$ then clearly we can simplify the previous agent to

[9] We recall that, given a substitution σ, the domain of σ is the finite set of variables $\{X \mid X\sigma \neq X\}$.

C[ask(true) → A + ask(d) → B][10]. In general, if a is the context constraint of C[ ], and for some constraint c′ we have that $\mathcal{D} \models \exists_{-\tilde{z}}\ a \wedge c \leftrightarrow a \wedge c'$ (where $\tilde{z} = Var(C, A)$), then we can replace c with c′. In particular, if we have that $a \wedge c$ is unsatisfiable, then c can immediately be replaced with false (the unsatisfiable constraint). In order to formalize this intuitive idea, we start with the following definition.

Definition 4. Let D be a (fixed) set of declarations, and s be a set of predicates. Given an agent A, its *weakest produced constraint* (w.r.t. s), denoted by $wpc_s(A)$, is defined by structural induction as follows:

$$\begin{aligned} wpc_s(stop) &= true \\ wpc_s(tell(c)) &= c \\ wpc_s(A \parallel B) &= wpc_s(A) \wedge wpc_s(B) \\ wpc_s(\textstyle\sum_i ask(c_i) \rightarrow A_i) &= true \\ wpc_s(p(\tilde{t})) &= \begin{cases} wpc_{(s \cup \{p\})}(A) & \text{if } p \notin s \text{ and } p(\tilde{t}) \leftarrow A \in defn_D(p(\tilde{t})) \\ true & \text{if } p \in s \end{cases} \end{aligned}$$

s contains then the set of predicates which should not be taken into consideration. Given a context C[ ] and a set of predicate symbols s the *weakest produced constraint*, of C[ ] (w.r.t. s) $wpc_s(C[\,])$, is inductively defined as follows:

$wpc_s([\,]) = true$
$wpc_s(C'[\,] \parallel B) = wpc_s(B) \wedge wpc_s(C'[\,])$
$wpc_s(\sum_{i=1}^{n} ask(c_i) \rightarrow A_i) = c_j \wedge wpc_s(C'[\,])$ where $j \in [1, n]$ and $A_j = C'[\,]$

Notice that the weakest produced constraint depends on the set of declarations D under consideration. We are now ready to define the operation of guard simplification.

Definition 5 (Guard Simplification). Let D be a set of declarations, and

$$d:\ H \leftarrow C[\textstyle\sum_{i=1}^{n} ask(c_i) \rightarrow A_i]$$

be a declaration of D. Assume that for some constraints $c'_1, \ldots, c'_n$ we have that for $j \in [1, n]$,

$\mathcal{D} \models \exists_{-\tilde{z}_j}\ wpc_\emptyset(C[\,]) \wedge c_j \leftrightarrow wpc_\emptyset(C[\,]) \wedge c'_j$ (where $\tilde{z}_j = Var(C, H, A_j)$), then

we can replace d with

$$d':\ H \leftarrow C[\textstyle\sum_{i=1}^{n} ask(c'_i) \rightarrow A_i] \qquad \square$$

In our AUCTION example, we can consider the weakest produced constraint of tell(LastBid ≠ quit), and modify the subsequent ask constructs as follows

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
      ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList'] ∧ LastBid ≠ quit ∧ HisBid = quit) →
         stop
   + ask(∃HisBid,HisList' [LastBid|As] = [HisBid|HisList']) →
         tell([LastBid|As] = [HisBid|HisList']) ||
         ...
```

[10] Note that in general the further simplification to C[A + ask(d) → B] is not correct, while we can transform C[ask(true) → A] into C[A].

Via the same operation, we can immediately simplify this to.

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || bidder_a(Bs, As) ||
      ask(false) → stop
   + ask(true) → tell([LastBid|As] = [HisBid|HisList']) ||
         ...
```

Branch Elimination and Conservative Guard Evaluation Notice that in the above program, we have a guard ask(false) which of course will never be satisfied. The first important application of the guard simplification operation regards then the elimination of unreachable branches.

Definition 6 (Branch elimination). Let

$$\mathsf{d}: \ \mathsf{H} \leftarrow \mathsf{C}[\textstyle\sum_{i=1}^{n} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}]$$

be a declaration. Assume that $n > 1$ and that for some $j \in [1, n]$, we have that $\mathsf{c_j} \equiv \mathsf{false}$, then we can replace d with

$$\mathsf{d}': \ \mathsf{H} \leftarrow \mathsf{C}[(\textstyle\sum_{i=1}^{j-1} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i}) \ + \ (\sum_{i=j+1}^{n} \mathsf{ask}(\mathsf{c_i}) \rightarrow \mathsf{A_i})] \qquad \square$$

The condition that $n > 1$ ensures that we are not eliminating all the branches (if we wanted to do so, and of course if we were allowed to, that is, if all the guards are unsatisfiable, then we could do so by replacing the whole choice with a new special agent, say dead whose semantics would be of always deadlocking, never affecting the constraint store).

By applying this operation to the above piece of example, we can eliminate ask(false) → stop, obtaining

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
   ask(true) → tell([LastBid|As] = [HisBid|HisList']) ||
         ...
```

Now we don't see any reason for not eliminating the guard ask(true) altogether. This can indeed be done via the following operation

Definition 7 (Conservative ask evaluation). Consider the declaration

$$\mathsf{d}: \ \mathsf{H} \leftarrow \mathsf{C}[\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{B}]$$

We can transform d into the declaration

$$\mathsf{d}': \ \mathsf{H} \leftarrow \mathsf{C}[\mathsf{B}] \qquad \square$$

This operation, although trivial, is subject of debate. In fact, Sahlin in [17] defines a similar operation, with the crucial distinction that the choice might still have more than one branch, in other words, in the system of [17] one is allowed to simplify the agent C[ask(true) → A + ask(b) → B] to the agent C[A], even if b is satisfiable. Ultimately, one is allowed to replace the agent C[ask(true) → A + ask(true) → B] either with C[A] or with C[B], indifferently. Such an operation is clearly more widely applicable than the one we have presented (hence the attribute "conservative" for

the operation we present) but is bound to be ***incomplete***, i.e. to lead to the lost of potentially successful branches. Nevertheless, Sahlin argues that an ask evaluation such as the one defined above is potentially too restrictive for a number of useful optimization. We agree with the statement only partially, nevertheless, the system we propose will eventually be equipped with a non-conservative guard evaluation operation as well (which of course, if employed, will lead to weaker correctness results). Such operation is, for space reasons, now omitted.

In our example program, the application of these branch elimination and conservative ask evaluation leads to the following:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
   tell([LastBid|As] = [HisBid|HisList']) ||
   make_new_bid_b(HisBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits")
   + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
         tell(MyBid ≠ quit) ||
         bidder_b(HisList',Bs')
```

Via a tell evaluation of tell([LastBid|As] = [HisBid|HisList']), this simplifies to:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) ||
   bidder_a(Bs, As) ||
   make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits")
   + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
         tell(MyBid ≠ quit) ||
         bidder_b(As,Bs')
```

3.4 Distribution

A crucial operation in our transformation system is the ***distribution***, which consists of bringing an agent inside a choice as follows: from the agent $A \parallel \sum_i \mathsf{ask}(c_i) \rightarrow B_i$, we want to obtain the agent $\sum_i \mathsf{ask}(c_i) \rightarrow (A \parallel B_i)$. This operation was introduced for the first time in the context of CLP in [7], and requires delicate applicability conditions, as it can easily introduce deadlock situation: consider for instance the following contrived program D.

```
p(Y) ← q(X) || (ask(X >= 0) → tell(Y=0))
q(0) ← stop
```

In this program, the process D.p(Y) originates the derivation $\langle \mathsf{D.p(Y)}, \mathsf{true} \rangle \rightarrow^* \langle \mathsf{D.stop}, \mathsf{Y} = 0 \rangle$. However, if we blindly apply the distribution operation to the first definition we would change D into:

```
p(Y) ← ask(X >= 0) → (q(X) || tell(Y=0))
```

and now we have that $\langle \mathsf{D.p(Y)}, \mathsf{true} \rangle$ generates only deadlocking derivations.

This situation is avoided by demanding that the agent being distributed will in any case not be able to produce any output before the choice is entered. This is done using the following notions of *required variable*. Recall that we denote by Stop any agent which contains only stop and || constructs.

Definition 8 (Required Variable). Let D.A be a process. We say that D.A *requires* the variable X iff, for each satisfiable constraint c such that $\mathcal{D} \models \exists_X c \leftrightarrow c$, $\langle D.A, c\rangle$ has at least one finite derivation and moreover $\langle D.A, c\rangle \rightarrow^* \langle D.A', c'\rangle$ implies that $\mathcal{D} \models \exists_{-\tilde{z}}\, c \leftrightarrow \exists_{-\tilde{z}}\, c'$, where $\tilde{z} = Var(A)$. □

In other words, the process D.A requires the variable X if, in the moment that the global store does not contain any information on X, then D.A cannot produce any information which affect the variables occurring in A and has at least one finite derivation. Even though the above notion is not decidable in general, in some cases it is easy to individuate required variables. For example it is immediate to see that, in our program, bidder_a(Bs, As) requires Bs: in fact the derivation starting in bidder_a(Bs, As) suspends (without having provided any output) after one step and resumes only when Bs has been instantiated. This example could be easily generalized. We can now give the formal definition of the distribution operation.

Definition 9 (Distribution). Consider a declaration

$$d:\ H \leftarrow C[A \parallel \sum_{i=1}^{n} ask(c_i) \rightarrow B_i]$$

The *distribution* of A in d yields as result the definition

$$d':\ H \leftarrow C[\sum_{i=1}^{n} ask(c_i) \rightarrow (A \parallel B_i)]$$

provided that A requires a variable which does not occur in H nor in C. □

The above applicability condition ensures that bringing A in the scope of the $ask(c_i)$'s will not introduce deadlocking derivations: In fact it is intuitively clear that the fact that A requires a variable X implies, by definition, that A can produce some output only in the moment that X is instantiated, but since X does not occur in H nor in C, we have that this can only happen once the choice is entered. Summarizing, the applicability conditions ensure that (in the initial definition) A might produce an output only after the choice is entered. This ensures that A cannot have an influence on the choice itself, and can be thus safely brought inside.

In our example, since the agent bidder_a(Bs, As) requires the variable Bs, which occurs only inside the ask guards, we can safely apply the distributive operation. The result is the following program.

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits") || bidder_a(Bs, As)
   + ask(MyBid ≠ quit) → tell(Bs = [MyBid|Bs']) ||
         tell(MyBid ≠ quit) ||
         bidder_a(Bs, As) ||
         bidder_b(As, Bs')
```

In this program we can now evaluate the construct tell(Bs = [MyBid|Bs']) obtaining (it is true that the variable Bs here occurs also elsewhere in the definition, but since it occurs only on choice-branches different than the one on which the considered agent lies, we can assume it to be renamed):

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits") || bidder_a(Bs, As)
```

```
  + ask(MyBid ≠ quit) → tell(MyBid ≠ quit) ||
      bidder_a([MyBid|Bs'], As) ||
      bidder_b(As, Bs')
```

Before we introduce the fold operation, let us clean up the program a bit further: by properly transforming the agent bidder_a(Bs, As) in the first ask branch, we easily obtain:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
     ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits") || stop
  + ask(MyBid ≠ quit) → tell(MyBid ≠ quit) ||
      bidder_a([MyBid|Bs'], As) ||
      bidder_b(As, Bs')
```

The just introduced stop agent can then safely be removed.

3.5 Folding

The folding operation has a special rôle in the panorama of the transformation operations. This is due to the fact that it allows to introduce recursion in a definition, often making it independent from the previous definitions. As previously mentioned, the applicability conditions that we use here for the folding operation do not depend on the transformation history, nevertheless, we require that the declarations used to fold an agent appear in the initial program. Thus, before defining the fold operation, we need the following.

Definition 10. A *transformation sequence* is a sequence of programs $D_0, \ldots, D_n$, in which D_0 is an *initial program* and each D_{i+1}, is obtained from D_i via one of the following transformation operations: definition introduction, unfolding, distribution, guard simplification, branch elimination, conservative guard evaluation and folding.

We also need the notion of *guarding context*. Intuitively, a context C[ ] is *guarding* if the "hole" appears in the scope of an ask guard[11]. Here $\equiv$ indicates syntactic equality.

Definition 11 (Guarding Context). A context C[ ] is a *guarding context* iff

$$C[\,] \equiv C'[\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i] \quad \text{and } A_j = C''[\,] \text{ for some } j \in [1, n]. \qquad \square$$

We can finally give the definition of folding:

Definition 12 (Folding). Let $D_0, \ldots, D_i$, $i \geq 0$, be a transformation sequence. Consider two definitions.

$$d : H \leftarrow C[A] \in D_i$$
$$f : B \leftarrow A \quad \in D_0$$

If C[ ] is a *guarding context* then *folding* A in d consists of replacing d by

$$d' : H \leftarrow C[B] \in D_{i+1}$$

(it is assumed here that d and f are suitably renamed so that the variable they have in common are only the ones occurring in A). $\square$

[11] Clearly, the scope of the ask guard in ask(c) → A is A.

The reach of this operation is best shown via our example. We can now fold auction_left(MyBid) in the above definition, and obtain:

```
auction_right(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_b(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("b quits")
   + ask(MyBid ≠ quit) → auction_left(MyBid)
```

Now, by performing an identical optimization on auction_left, we can also obtain:

```
auction_left(LastBid) ← tell(LastBid ≠ quit) || make_new_bid_a(LastBid,MyBid) ||
      ask(MyBid = quit) → tell(Bs = [quit|Bs']) || broadcast("a quits")
   + ask(MyBid ≠ quit) → auction_right(MyBid)
```

This part of the transformation shows in a striking way one of the main benefits of the folding operation: the saving of synchronization points. Notice that in the initial program the two bidders had to "wait" for each other. In principle they were working in parallel, but in practice they were always acting sequentially, since one always had to wait for the bid of the competitor. The transformation allowed us to discover this sequentiality and to obtain an equivalent program in which the sequentiality is exploited to eliminate all suspension points, which are known to be one of the major overhead sources. Furthermore, the transformation allows a drastic save of computational *space*. Notice that in the initial definition the parallel composition of the two bidders leads to the construction of two lists containing all the bids done so far. After the transformation we have a definition which does not build the list any longer, and which, by exploiting a straightforward optimization can employ only *constant* space.

4 Correctness

Any transformation system must be useful (i.e. allow useful transformations and optimization) and – most importantly – *correct*, i.e., it must guarantee that the resulting program is in some sense equivalent to the one we have started with. Having at hand a formal semantics for our paradigm, we defines *correctness* as follows.

Definition 13 (Correctness). A transformation sequence $D_0, \ldots, D_n$ is called

- *partially correct* iff for each agent A we have that $\mathcal{O}(D_0.A) \supseteq \mathcal{O}(D_n.A)$
- *complete* iff for each agent A we have that $\mathcal{O}(D_0.A) \subseteq \mathcal{O}(D_n.A)$
- *totally correct* iff it is both partially correct and complete. □

So a transformation is *partially correct* iff nothing is added to the semantics of the initial program and is *complete* iff no semantic information is lost during the transformation. We can now state the main result of this paper.

Theorem 14 (Total Correctness). Let $D_0, \ldots, D_n$ be a transformation sequence. Then $D_0, \ldots, D_n$ is *totally correct.* □

This theorem is originally inspired by the one of Tamaki and Sato for pure logic programs [22], and has retained some of its notation. Of course the similarities don't go much further, as demonstrated by the fact that in our transformation system the

applicability conditions of folding operation do not depend on the transformation history (while allowing the introduction of recursion), and that the folding definitions are allowed to be recursive (the distinction between P_{new} and P_{old} of [22] is now superfluous).

It is important to notice that – given the definition of observable we are adopting (Definition 1) – the initial program D_0 and the final one D_n have exactly the same successful derivation and the same deadlocked derivation. The first feature (regarding successful derivations) is to some extent the one we expect and require from a transformation, because it corresponds to the intuition that D_n "produces the same results" of D_0. Nevertheless, also the second feature (preservation of deadlock derivation) has an important rôle. Firstly, it ensures that the transformation does not introduce deadlock point, which is of crucial importance when we are using the transformation for optimizing a program. Secondly, this feature allows to use the transformation as a tool for proving deadlock freeness (i.e., absence of deadlock). In fact, if, after the transformation we can prove or or see that the process $D_n.A$ does never deadlock, then we are also sure that $D_0.A$ does not deadlock either.

5 Related Work

In the literature, there exist three paper which are relatively closely related to the present one: de Francesco and Santone's [8], Ueda and Furukawa's [23], and Sahlin's [17]: in [8] it is presented a transformation system for CCS [14], in [23] it is defined a transformation system for Guarded Horn Clauses, while in [17] it is presented a transformation system for AKL.

Common to all three cases is that our proposal improves on them by introducing new operations such as the distribution, the techniques for the simplification of constraint, branch elimination and conservative guard evaluation (though, some constraint simplification is done in [17] as well). Because of this, the transformation system we are proposing can be regarded as an extension of the ones in the paper above. Notice that without the above-mentioned operations the transformation of our example would not be possible. Further, we provide a more flexible definition for the folding operation, which allows the folding clause to be recursive, and frees the *initial program* from having to be partitioned in P_{new} and P_{old}.

Other minor differences between our paper and the [23, 17] are the following ones. Compared to [23], our systems takes advantage of the greater flexibility of the CCP (wrt GHC). For instance, we can define the unfolding as a simple body replacement operation without any additional applicability condition, while this is not the case for GHC. Going on to [17], an interesting difference between it and this paper which is worth remarking is the one we have already mentioned in the discussion after Definition 7: in [17] it is considered a definition of *ask evaluation* which allows to remove potentially selectable branches; the consequence is that the resulting transformation system is only *partially* (thus not totally) correct. However, we should mention that in [17] two preliminary assumptions on the "scheduling" are made in such a way that this limitation is actually less constraining that it might appear. In any case, as we already said, the extended version of this transformation system will encompass an operation of *non-conservative* guard expansion, analogous to the one of [17] (and which – if employed – will necessarily lead to weaker correctness results).

Concluding, we want to mention that a previous work of the authors on the subject is [7] which focuses primarily on CLP paradigm (with dynamic scheduling), and is concerned with the preservation of deadlock derivation along a transformation. In [7], for the first time, it was employed a transformation system in order to prove absence of deadlock of a program (HAMMING). The second part of [7] contains a sketch of a primitive version of an unfold/fold transformation for CCP programs. Nevertheless, the system we are presenting here is (not only much more extended, but also) different in nature from [7]. This is clear if one compares the definitions of folding, which, it is worth reminding, is *the* central operation in an Unfold/Fold transformation system. In [7] this operation requires severe constraints on the initial program and applicability conditions which rely on the *transformation history*, while here the only requirement is that the folding has to take place inside a guarding context, which is a plain syntactic condition. As a consequence we have that

– This system is – generally speaking – of much broader applicability.

All limitations on the initial programs are dropped. Ultimately, the folding definition is allowed to be recursive (which is really a step forward in the context of folding operations which are themselves capable of introducing recursion). Of course – being the two systems of different nature – one can invent an example transformation which is doable with the tools of [7] but not with the ones here presented. We strongly believe that such cases regard contrived examples of no practical relevances.

– The folding operation presented here is much simpler.

This is of relevance given the fact that the complexity of applicability of the folding operation has always been one of the major obstacle both in implementing it and in making it accessible to a wider audience.

In particular, as opposed to virtually all fold operations which enable to introduce recursion presented so far (the only exception being [8]), the applicability of the folding operation does not depend on the transformation history, (which has always been one of the "obscure sides" of it) but it relies on plain syntactic criteria.

We also should mention that because of the structural differences, the proofs for this paper are necessarily completely different.

Moreover, we have introduced new operations. In particular the guard simplification (which brings along the *branch elimination* and the *conservative guard evaluation*) is of crucial importance in order to have a transformation system which allows fruitful optimizations. Concluding, another fundamental operation for CCP – the distributive operation – has now simpler applicability conditions, which help in checking it in a much more straightforward way.

References

1. N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.
2. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
3. K.L. Clark and S. Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of IJCAI'77*, pages 419–120, 1977.

4. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 1998. to appear.
5. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1):101–146, 1996.
6. S. Etalle and M. Gabbrielli. Partial evaluation of concurrent constraint languages. *ACM Computing Surveys*, 1998. to appear.
7. S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *ACM-SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulation*. ACM Press, 1997.
8. N. De Francesco and A. Santone. Unfold/fold transformation of concurrent processes. In H. Kuchen and S.Doaitse Swierstra, editors, *Proc. 8th Int'l Symp. on Programming Languages: Implementations, Logics and Programs*, volume 1140, pages 167–181. Springer-Verlag, 1996.
9. C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.
10. N. Jørgensen, K. Marriot, and S. Michaylov. Some Global Compile-Time Optimizations for CLP($\mathcal{R}$). In *Proc. 1991 Int'l Symposium on Logic Programming*, pages 420–434, 1991.
11. H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Proc. Ninth ACM Symposium on Principles of Programming Languages*, pages 255–267. ACM, 1982.
12. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.
13. M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, March 1993.
14. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
15. T Mogensen and P Sestoft. Partial evaluation. In A. Kent and J.G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. M. Dekker, 1997.
16. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
17. D. Sahlin. Partial Evaluation of AKL. In *Proceedings of the First International Conference on Concurrent Constraint Programming*, 1995.
18. V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989.
19. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.
20. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 1991.
21. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS. Springer-Verlag, 1995. see www.ps.uni-sb.de/oz/.
22. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
23. K. Ueda and K. Furukawa. Transformation rules for GHC Programs. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 582–591. Institute for New Generation Computer Technology, Tokyo, 1988.