Guarded Constructive Disjunction:
Angel or Demon?

Christian Codognet
LIENS / University of Paris XIII
45 rue d'Ulm, 75005 Paris, FRANCE
Christian.Codognet@ens.fr

Philippe Codognet
INRIA-Rocquencourt
B. P. 105, 78153 Le Chesnay, France
Philippe.Codognet@inria.fr

Quoniam sepe, ut mihi videtur, expertus sum, quod cum rationes viderem ad utramque partem probabiles, tamen ad neutram partem iudicii determinabam me (...) sed in suspenso tenebam me.

Jean Buridan. Quaestiones super decem libros Ethicorum.

Jean Buridan (1300-1358) studied logic, ethics and physics. He taught at the university of Paris. (...) Buridan was known as a student of Ockham and was an intellectual determinist: in choosing one of several possible decisions, the “will” is controlled by reason. It acts only when the reason decides that one of the possibilities is the best (the will also chooses this possibility). But if reason decides that several possibilities are equivalent, the will does not act. This is the source of the well-known example of “Buridan's ass”, which died while standing between two identical handfuls of grass.

N. I. Styazhkin. History of Mathematical Logic from Leibniz to Peano.

Abstract. We propose a new operator for Concurrent Constraint (cc) programming, called guarded constructive disjunction, in order to avoid indeterminism, aka demonic non-determinism, aka don’t care non-determinism. This operator is deterministic, avoiding thus confluence problems, and extends constructive disjunction by guarding each alternative branch. Our framework is based on a denotational semantics of concurrent constraint languages, where each agent, including the guarded constructive disjunction operator, is seen as a closure operator over the lattice defined by the constraint system. We investigate the interest of this semantics for program analysis in the abstract interpretation paradigm, thanks to the notion of abstraction between constraint systems. We illustrate with some programming examples how suspension analysis can be performed within this framework.
1 Introduction

Concurrent Constraint (cc) languages have been proposed originally by [23] [24]; they extend Constraint Logic Programming (CLP) with concurrency features that have emerged from the domain of Concurrent Logic Programming. The key idea underlying the family of cc languages is to use constraints to extend the synchronization and control mechanisms of concurrent logic languages. Briefly, multiple agents (processes) run concurrently and interact by means of a shared store, i.e. a global set of constraints. Each agent can either add a new constraint to the store (Tell operation) or check whether or not the store entails a given constraint (Ask operation). Synchronization is achieved through a blocking ask: an asking agent may block and suspend if one cannot state if the asked constraint is entailed nor its negation is entailed. Thus nothing prevents the constraint to be entailed later in the execution, but the store does not yet contain enough information. The agent is suspended until other concurrently running agents add (Tell) new constraints to the store to make it strong enough to decide.

The general cc framework makes room for both angelic non-determinism and demonic non-determinism (indeterminism). However if angelic nondeterminism causes no problem from the semantical point of view, indeterminism is more problematic to incorporate because it gives rise to non-confluent semantics. Observe also that indeterminism requires, from a pragmatic programming point of view, guarding each alternative with some condition, as originally proposed by [11]. Non-confluency is an important problem and particularly hampers the design of program analysis frameworks. It notably prevents, together with the problem of Ask approximation [31, 32], to directly lift CLP analysis frameworks to cc analysis. It seems therefore natural to base an analysis framework on confluent semantics. This kind of approach has been taken in [13], which nevertheless only leads to the definition of an obvious class of indeterministic cc programs ensured to be confluent: when for all indeterministic choices either all guards are mutually exclusive or all guards are reduced to true. Our approach is also to propose a confluent semantics and, in order to avoid indeterminism, we choose to design a new operator for cc languages called guarded constructive disjunction (GCD). This operator is deterministic, thus avoiding confluence problems, and extends constructive disjunction by guarding each alternative branch. We therefore move from an irreversible choice construct (i.e. indeterminism) to a refinable non-choice construct (i.e. GCD operator). The interest of defining a new operator in the language makes it possible to analyze a larger class of programs than in [13], because non mutually-exclusive guards can be handled and common information extracted from the corresponding branches.

Indeed, the GCD operator generalizes the \( \nabla \) operator, a particular form of constructive disjunction, introduced in [30] in the case of finite domain constraints. Note that constructive disjunction is very close to the generalized propagation method of [19], which attempts, in the constraint logic programming framework, to extract common information from disjunctive branches. The interest of constructive disjunction for improving efficiency in constraint logic programming with disjunctive constraints has been assessed by [16] and [20]. The