Explaining Constraint Programming

Krzysztof R. Apt[1,2,3]

[1] School of Computing, National University of Singapore
[2] CWI, Amsterdam
[3] University of Amsterdam, The Netherlands

Abstract. We discuss here constraint programming (CP) by using a proof-theoretic perspective. To this end we identify three levels of abstraction. Each level sheds light on the essence of CP.

In particular, the highest level allows us to bring CP closer to the computation as deduction paradigm. At the middle level we can explain various constraint propagation algorithms. Finally, at the lowest level we can address the issue of automatic generation and optimization of the constraint propagation algorithms.

1 Introduction

Constraint programming is an alternative approach to programming which consists of modelling the problem as a set of requirements (constraints) that are subsequently solved by means of general and domain specific methods.

Historically, constraint programming is an outcome of a long process that has started in the seventies, when the seminal works of Waltz and others on computer vision (see, e.g.,[30]) led to identification of constraint satisfaction problems as an area of Artificial Intelligence. In this area several fundamental techniques, including constraint propagation and enhanced forms of search have been developed.

In the eighties, starting with the seminal works of Colmerauer (see, e.g., [16]) and Jaffar and Lassez (see [21]) the area constraint logic programming was founded. In the nineties a number of alternative approaches to constraint programming were realized, in particular in ILOG solver, see e.g., [20], that is based on modeling the constraint satisfaction problems in C++ using classes. Another, recent, example is the Koalog Constraint Solver, see [23], realized as a Java library.

This way constraint programming eventually emerged as a distinctive approach to programming. In this paper we try to clarify this programming style and to assess it using a proof-theoretic perspective considered at various levels of abstraction. We believe that this presentation of constraint programming allows us to more easily compare it with other programming styles and to isolate its salient features.

2 Preliminaries

Let us start by introducing the already mentioned concept of a constraint satisfaction problem. Consider a sequence $X = x_1, \ldots, x_m$ of variables with respective

A. Middeldorp et al. (Eds.): Processes... (Klop Festschrift), LNCS 3838, pp. 55–69, 2005.

domains $D_1, \ldots, D_n$. By a ***constraint*** on X we mean a subset of $D_1 \times \ldots \times D_m$. A ***constraint satisfaction problem (CSP)*** consists of a finite sequence of variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$ and a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write such a CSP as

$$\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle.$$

A ***solution*** to a CSP is an assignment of values to its variables from their domains that satisfies all constraints. We say that a CSP is ***consistent*** if it has a solution, ***solved*** if each assignment is a solution, and ***failed*** if either a variable domain is empty or a constraint is empty. Intuitively, a failed CSP is one that obviously does not have any solution. In contrast, it is not obvious at all to verify whether a CSP is solved. So we introduce an imprecise concept of a '***manifestly solved***' CSP which means that it is computationally straightforward to verify that the CSP is solved. So this notion depends on what we assume as 'computationally straightforward'.

In practice the constraints are written in a first-order language. They are then atomic formulas or simple combinations of atomic formulas. One identifies then a constraint with its syntactic description. In what follows we study CSPs with finite domains.

3 High Level

At the highest level of abstraction constraint programming can be seen as a task of formulating specifications as a CSP and of solving it. The most common approach to solving a CSP is based on a ***top-down search*** combined with ***constraint propagation***.

The top-down search is determined by a *splitting strategy* that controls the splitting of a given CSP into two or more CSPs, the 'union' of which (defined in the natural sense) is ***equivalent*** to (i.e, has the same solutions as) the initial CSP. In the most common form of splitting a variable is selected and its domain is partitioned into two or more parts. The splitting strategy then determines which variable is to be selected and how its domain is to be split.

In turn, constraint propagation transforms a given CSP into one that is equivalent but *simpler*, i.e, easier to solve. Each form of constraint propagation determines a notion of ***local consistency*** that in a loose sense approximates the notion of consistency and is computationally efficient to achieve. This process leads to a search tree in which constraint propagation is alternated with splitting, see Figure 1.

So the nodes in the tree are CSPs with the root (level 0) being the original CSP. At the even levels the constraint propagation is applied to the current CSP. This yields exactly one direct descendant. At the odd levels splitting is applied to the current CSP. This yields more than one descendant. The leaves of the tree are CSPs that are either failed or manifestly solved. So from the leaves of the trees it is straightforward to collect all the solutions to the original CSP.

The process of tree generation can be expressed by means of proof rules that are used to express transformations of CSPs. In general we have two types of