A Conservative Approach to Meta-Programming in Constraint Logic Programming

Pierre Lim and Joachim Schimpf

European Computer-Industry Research Centre
Arabellstraße 17, 81925 München, Germany
{pierre,joachim}@ecrc.de

Abstract. Constraint Logic Programming [4] extends Logic Programming by generalizing the notion of unification to constraint solving. This is achieved by fixing the interpretation of some of the symbols in the language. The two alternative mechanisms used in the currently implemented CLP systems to achieve this operation are: (1) fix the interpretation before the program executes or (2) fix the interpretation at a point during program execution when it is used in a constraint. CLP(∀) [5] and Prolog-III [1] take the first approach whereas CHIP [2] takes the second approach. The problem with the first approach is that interpreted terms cannot be manipulated syntactically. The problem with the second approach is that all constraint operations have to be made explicit and this increases the difficulty of programming. We propose a synthesis of both approaches that overcomes their individual difficulties. Our method is implemented in the ECLiPS® compiler system.

1 Introduction

The fundamental operation of unification in Logic Programming (LP) has been generalized to constraint solving in Constraint Logic Programming (CLP) [4]. Although this generalization greatly improves the efficiency and utility of CLP languages compared to LP languages it also complicates meta-programming. The problem is to decide how and when to assign the fixed interpretations of some of the functors. For example, the functors 1, 2 and + in an arithmetic CLP language are interpreted respectively as the arithmetic constants one, two and the addition function. So the equation 1 + 2 = X + Y is equivalent to 3 = X + Y. However, for meta-programming the symbols 1, 2 and + should be treated simply as uninterpreted symbols, so that the equation 1 + 2 = X + Y has the solution { X = 1, Y = 2 }. It is not equivalent to 3 = X + Y which is unsatisfiable. The reconciliation of this overloading of functors is addressed by Heintze et al. [3] in which they give a theoretical framework for the problem and discuss a solution for the CLP(∀) language. The problem with their method is that it is not conservative i.e. it does not preserve the current LP meta-programming functionality, but rather it defines new functionality to replace that which was lost. The conservation of current functionality is important because it means
that tools, techniques and applications developed for LP systems are usable on
CLP systems. On the other hand, CHIP which distinguishes constraints syntac-
tically has no problem with meta-programming but every constraint operation
has to be made explicit, i.e. all head unifications are syntactic not semantic. This
is counter-intuitive if one expected, say, the + symbol to denote addition. Moreover
the requirement for explicit constraint operations places an extra burden
on the programmer.

We present a simple syntactic transformation which achieves a synthesis
of both approaches and overcomes their individual difficulties and provide an
implementation in the ECLiPSe\textsuperscript{1} system. Our presentation is organized in the
following way. First, we define the class of structures we are dealing with, i.e.
those containing uninterpreted functors. The extensions to unification required
by CLP are then discussed. Next, the approach of [3] is briefly reviewed. We use
their theoretical basis in further discussions of the meta-programming problem
and the solution. The CHIP approach is then discussed and be build on this ap-
proach to develop our solution. Our solution and its implementation in ECLiPSe
is then given. In sections 7 and 8 we present a comparison with the approach of
[3] and give our solutions to their examples. Finally some concluding remarks
are made and a summary of our results is given.

2 Structures with uninterpreted functors

The fundamental extension of LP to CLP is the assignment of a non-Herbrand
interpretation to some of the function symbols in the language and the inclu-
sion of relations other than syntactic equality (according to a given algebraic
description called the structure of computation). Of particular importance is
the structure of the Herbrand Universe (HU) since this is the core of the Prolog
programming language. In order to utilize Prolog programming techniques un-
interpreted functors have to be included. We define the class of structures with
uninterpreted functors which we denote parametrically as HU(\mathcal{D}) where \mathcal{D}
represents the underlying algebraic structure e.g. rationals, reals, finite domains.
Prolog has the structure HU(\bot) since there is no structure under that of the
uninterpreted functors.

We now give some definitions and then proceed to consider the types in
these structures. A sort is a name of a type and a signature is a sequence of sorts.
The alphabet of a CLP(HU(\mathcal{D})) language is partitioned into several classes.

- \Pi is the set of uninterpreted (programmed) predicate symbols, e.g. laplace,
fibonacci, nqueens.
- \Pi_D \setminus \Pi = \emptyset is the set of interpreted predicate symbols and contains
  at least = (syntactic equality) in addition to any other predicates in \mathcal{D}, e.g.
  for the rational arithmetic structure in CHIP the following symbols denote

\textsuperscript{1} ECLiPSe is the platform on which work on constraint handling is being performed
at ECRC.