Compiling Constraint Handling Rules into Prolog with Attributed Variables

Christian Holzbaur\(^1\) and Thom Frühwirth\(^2\)

\(^1\) University of Vienna
Department of Medical Cybernetics and Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria
christian@ai.univie.ac.at

\(^2\) Ludwig-Maximilians-University
Department of Computer Science
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de

Abstract. We introduce the most recent and advanced implementation of constraint handling rules (CHR) in a logic programming language, which improves both on previous implementations (in terms of completeness, flexibility and efficiency) and on the principles that should guide such a Prolog implementation consisting of a runtime system and a compiler. The runtime system utilizes attributed variables for the realization of the constraint store with efficient retrieval and update mechanisms. Rules describing the interactions between constraints are compiled into Prolog clauses by a multi-phase compiler, the core of which comprises a small number of compact code generating templates in the form of definite clause grammar rules.

Keywords: Logic and constraint programming, Implementation and compilation methods.

1 Introduction

In the beginning of constraint logic programming (CLP), constraint solving was “hard-wired” in a built-in constraint solver written in a low-level language. While efficient, this so-called “black-box” approach makes it hard to modify a solver or build a solver over a new domain, let alone debug, reason about and analyze it. This is a problem, since one lesson learned from practical applications is that constraints are often heterogeneous and application-specific. Consequently, several proposals have been made to allow more for flexibility and customization of constraint systems (“glass-box” or even “no-box” approaches):

- Demons, forward rules and conditionals in CHIP [6] allow the definition of propagation of constraints in a limited way.

\(^*\) Part of this work was performed while visiting CWG at LMU with financial support from DFG.
– Constraint combinators in cc(FD) [13] allow to build more complex constraints from simpler constraints.
– Constraints connected to a Boolean variable in BNR-Prolog [2] and “nested constraints” [31] allow to express any logical formula over primitive constraints.
– Indexicals in clp(FD) [5] allow to implement constraints over finite domains at a medium level of abstraction.
– Meta- and attributed variables [26], [21], [15] allow to attach constraints to variables at a low level of abstraction.

It should be noted that all the approaches but the last can only extend a solver over a given, specific constraint domain, typically finite domains. The expressive power to realize other (application-specific) constraint domains is only provided by the last approach.

Attributed variables provide direct access storage locations for properties associated with variables. When such variables are unified, their attributes have to be manipulated. Thus attributed variables make unification user-definable [15], [16], [17]. Attributed variables require roughly the same implementation effort as hard-wired delay (suspension) and coroutining mechanisms found in earlier Prolog implementations, while being more general. And indeed, attributed variables nowadays serve as the primary low-level construct for implementing suspension (delay) mechanisms and constraint solver extensions in many constraint logic programming languages, e.g. SICStus [4] and ECLiPSe [3] Prolog. However writing constraints this way is tedious, a kind of “constraint assembler” programming.

If there already is a powerful constraint assembler, one may wonder what an associated high-level language could look like. Our proposal is a declarative language extension especially designed for writing constraint solvers, called constraint handling rules (CHR) [10], [12], [18], [11]. With CHR, one can introduce user-defined constraints into a given high level host language, be it Prolog or Lisp. As language extension, CHR themselves are only concerned with constraints, all auxiliary computations are performed in the host language. CHR have been used in dozens of projects worldwide to encode dozens of constraint handlers (solvers), including new domains such as terminological and temporal reasoning. If comparable hard-wired constraint solvers are available, the price to pay for the flexibility of CHR is often within an order of magnitude in runtime. The performance gap can in many cases be eliminated by tailoring the CHR constraints to the specifics of the class of applications at hand.

CHR is essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR can define both simplification of and propagation over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification. CHR can be seen as a generalization of the various CHIP [6] constructs for user-defined constraints.