Entailment Simplification and Constraint Constructors for User-Defined Constraints*

Thom Frühwirth
ECRC, Arabellastrasse 17, D-8000 Munich 81, Germany
thom@ecrc.de

January 10, 1995

Abstract. We investigate how to implement entailment simplification and, more generally, constraint constructors for user-defined constraints. Entailment simplification was introduced, for feature terms, by Ait-Kaci, Podelski and Smolka, and constraint constructors include the implication operator (Saraswat), the cardinality operator (Van Hentenryck), the choice statements of AKL (Haridi et al.) and the conditional of OZ (Smolka). We assume that user-defined constraints are defined by constraint handlers written in a CLP language extended with constraint handling rules (Fruehwirth). The idea is to get entailment simplification for free from given constraint handlers by extending the operational semantics of constraint handling rules and to implement constraint constructors with constraint handling rules. We also propose a generic constraint constructor called guarded disjunction.

1 Introduction

Constraint handling rules (CHRs) [Fru92] are a *language extension* providing the user (application-programmer) with a declarative and flexible means to introduce *user-defined* constraints (in addition to *built-in* constraints of the underlying language). CHRs are essentially multi-headed guarded rules. CHRs define *simplification* of and *propagation* over user-defined constraints. Simplification replaces conjunctions of constraints by simpler ones while preserving logical equivalence. Propagation adds new constraints which are logically redundant (but may cause further simplification). When repeatedly applied the constraints become simplified and may become solved. In this way, a set of CHRs defines a constraint handler. If a constraint handler always solves the constraints, we call it a constraint solver.

*Part of this work is supported by ESPRIT Project 5291 CHIC

User-defined constraint handling is a very active area of research. CHIP was the first constraint logic programming language to introduce the necessary constructs (demons, forward rules, conditionals) [D*88]. These various constructs have been generalised into CHRs. Constraints are seen as a computationally efficient incarnation of the predicates defined in the underlying host language. CHRs have a logical reading and thus preserve the declarative semantics of the underlying logic programming language they extend. Thus we can reason about correctness, termination and confluence of a set of CHRs. The representation of constraints in the same formalism as the rest of the program greatly facilitates the prototyping, extension, specialization and combination of constraint handlers.

Constraint entailment was introduced to give declarative semantics to [Mah87] and to synchronise concurrent execution of guarded rules in concurrent logic programming [Sha89]. At the same time it was used in CLP languages to allow for more powerful programs [D*88, **?**]. The problem is to check if a conjunction of constraints (the context) implies (entails) another conjunction of constraints (the local constraint). Like constraint solving in traditional CLP languages, entailment checking should be *incremental*. This idea seemed to appear first in [APS92], where the incremental entailment checking of feature term constraints is called *entailment simplification*.

Constraint constructors we call operators over constraints especially designed to enable the user to build complex constraints from simpler ones. This idea has been first formulated by Van Hentenryck, who also proposed one such constructor, the cardinality operator [VH91]. Constraint constructors are often based on logical connectives (e.g. conditional) or can have the flavor of meta-predicates (e.g. cardinality).

In this work we show that already existing constraint handlers written in a CLP language with CHRscan be used to get entailment simplification for free. Then we consider some constraint constructors proposed in the literature, which all rely on constraint entailment as the basic operation, and investigate how they can be implemented by using CHRs with entailment simplification.

2 Some Constraint Constructors

Let C be a context (the conjunction of constraints in the constraint store), c_i's be local conjunctions of constraints, and a_i's,b be non-constraint atoms. Let $\rhd$ stand for a commitment operator, which is either $\rightarrow$ for don't care nondeterminism as in committed-choice languages or $\Rightarrow$ for don't know nondeterminism a la Prolog (adapting notation of [**?**]).

Various *conditionals* have been proposed in the literature. The *implication* constraint constructor proposed by [**?**], written $c \Longrightarrow a$, executes a as soon as c is entailed by the context C. If $\neg c$ is entailed, then the constructor simply succeeds, otherwise the constructor delays (flounders). The *if-then-else* constructor

of CHIP [D*88], written `if` c `then` a `else` b, behaves like the implication constructor, but in addition executes b if $\neg c$ is entailed. The conditional of Smolkas OZ language generalizes this to `if` c_1 `then` a_1 `[]` ... `[]` c_n `then` a_n `else` b.

We propose here a generic constraint constructor, called *guarded disjunction*, written $c_1 \rhd a_1; c_2 \rhd a_2; \ldots; c_n \rhd a_n$. If c_i is entailed, then $c_i \rhd a_i$ is removed from the guarded disjunction and the corresponding a_i is executed. If $\rhd$ is $\rightarrow$, the residual guarded disjunction succeeds. If $\rhd$ is $\Rightarrow$, it is re-activated on backtracking if a_i failed. If a $\neg c_i$ is entailed, the disjunct $c_i \rhd a_i$ is simply removed. A guarded disjunction with one disjunct only $c \Rightarrow a$ is replaced by a conjunction $c \wedge a$. An empty guarded disjunction corresponds to failure. Otherwise the constructor delays. Guarded disjunction is useful for executing concurrent committed-choice LP languages as well as CLP languages, as the completion of a predicate defined by clauses $p(X_1, \ldots, X_m) \leftarrow c_i \rhd a_i$ is the guarded disjunction $p(X_1, \ldots, X_m) \leftarrow c_1 \rhd a_1; c_2 \rhd a_2; \ldots; c_n \rhd a_n$, which can be unfolded deterministically. After proposing guarded disjunction, we heard of Oz, which relies on similar ideas using two constructors, the above-mentioned conditional and a deterministic disjunction.

In [VH91], a powerful constructor, the *cardinality operator* is described, written #(l,u,$[c_1, \ldots, c_n]$) ($l \leq u$) which succeeds if between l and u local constraints c_i are entailed by the context. If a c_i is entailed, the bounds l and u are decremented by one and the c_i is dropped from the list. If a $\neg c_i$ is entailed, the c_i is dropped from the list. If ($l \leq 0, n \leq u$) then the constructor succeeds. If ($n = l$) then all c_i must be entailed, hence we can replace the constructor by the conjunction of the c_i. Similarly, if ($u = 0$), we replace the constructor by the conjunction of all negated c_i. Otherwise the constructor delays.

3 Implementation

To be able to implement these constructors for user-defined constraints, we first need entailment simplification for a context C and a given set of c_i's.

Entailment Simplification. To represent the c_i's, we need either local constraint stores or to *index* the local constraints (e.g. by preprocessing them to have an additional argument). We choose the latter alternative, because it can be accomplished with a simple extension of the current prototype interpreter for constraint handling rules. As said earlier, we want to reuse already existing constraint handlers defined with CHRs. Note that propagation CHRs already define an entailment relation and the simplification CHRs define an equivalence relation.

The idea is to apply CHRs to the context and the local constraints while taking care of correctness. In a multi-headed CHR if all the heads match atomic constraints from the context (resp. from one local constraint), we add the body constraint of the CHR to the context (resp. local constraint by indexing it) as

usual. Clearly the heads should not match atomic constraints from different local constraints. If the heads match match atomic constraints from the context and from one local constraint, we don't touch the context - we do not remove any context constraint and we add the body constraint to local constraint.

Assume now we simplify a local constraint c_i. When is it entailed? Clearly, if c_i has been simplified to $\texttt{true}_i$, it is entailed. Analogously, if the simplification results in $\texttt{false}_i$, $\neg c_i$ is entailed. Moreover, a local constraint c_i is also entailed if all of its atomic constraints are also present in the context. In this case, we can simplify the local constraint to $\texttt{true}_i$.

Taking indexed constraints into account as described provides us with entailment simplification once and for all, without having to write constraint-specific entailment code.

Example. We illustrate entailment simplification with a user-defined constraint $\leq$.

```
(1a) X≤Y <=> X=Y | true.  % reflexivity
(1b) X≤Y,Y≤X <=> X=Y. % identity
(1c) X≤Y,Y≤Z ==> X≤Z. % transitivity
```

Simplification CHR (1a) states that `X≤X` is logically true. Hence, whenever we see the goal `X≤X` we can simplify it to `true`. Similarly, simplification CHR (1b) means that if we find `X≤Y` as well as `Y≤X` in the current goal, we can replace it by the logically equivalent `X=Y`. Propagation CHR (1c) states that the conjunction `X≤Y,Y≤Z` implies `X≤Z`.

The following example illustrates how the constraint handler works:

```
:-  A≤B,C≤A,B≤C.
% C≤A,A≤B propagates C≤B by 1c.
% C≤B,B≤C simplifies to B=C by 1b.
% C≤A,A≤B simplifies to A=B by 1b as C=B.
     A=B,B=C.
```

Now examples for entailment simplification (local constraints are indexed):

```
:-  A≤B,B≤A,B=C, A≤1C, A=2B.
% A≤B,B≤A simplifies to A=B by 1b.
% A≤1C simplifies to true1 by 1a as A=B,B=C.
% A=2B simplifies to true2 as A=B is in the context.
     A=B,B=C,true1,true2.
:-  A≤B, C≤1A, B≤C.
% C≤1A,A≤B propagates C≤1B by 1c.
% C≤1B,B≤C simplifies to B=1C by 1b.
% C≤1A,A≤B simplifies to A=1B by 1b as C=B.
% A≤B,B≤C propagates A≤C by 1c.
     A≤B,B≤C,A≤C, A=1B,B=1C.
```

Constraint Constructors. Whenever a c_i occuring to a constraint constructor is simplified to $\texttt{true}_i$ or $\texttt{fail}_i$, the constraint constructor reacts.

As an example, we implement the cardinality operator.

```
% Initialise - Call Indexed Local Constraints
#(L,U,Constraints) <=> length(Constraints,N), L=<U,0=<U,L=<N,
  call_uniquely_indexed_constraints(Constraints,Indices),
  #(L,U,N,Indices).
% Special Cases of Bounds
#(L,U,N,IL) <=> L=<0,N=<U | true.
#(L,U,N,IL) <=> N=<L | N=L,call_positive(IL).
#(L,U,N,IL) <=> U=<0 | U=0,call_negative(IL).
% Local Constraint Done
#(L,U,N,IL),true(I) <=> delete(I,IL,IL1),#(L-1,U-1,N-1,IL1).
#(L,U,N,IL),false(I) <=> delete(I,IL,IL1),#(L,U,N-1,IL1).
```

The first propagation CHR adds a conjunction of the indexed constraints, the other simplification CHRs define the behaviour of the constraint constructor according to its definition.

References

[APS92] H. Ait-Kaci, A. Podelski and G. Smolka, A Feature-Based Constraint System for Logic Programming with Entailment, Fifth Generation Computer Systems, Tokyo, Japan, June 1992.

[D*88] M. Dincbas et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.

[Fru92] T. Frühwirth, Constraint Simplification Rules (later renamed into CHRs), Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992 (revised version of internal report ECRC-91-18i, October 1991).

[H*93] S. Haridi et al., Concurrent Constraint Programming at SICS with the Andorra Kernel Language, First Workshop on Principles and Practice of Constraint Programming, Newport, RI, USA, April 28-30, 1993.

[Mah87] Maher M. J., Logic Semantics for a Class of Committed-Choice Programs, Fourth Intl Conf on Logic Programming, Melbourne, Australia, MIT Press, pp 858-876.

[Sar93] V. A. Saraswat, Concurrent Constraint Programming Languages, MIT Press, 1993.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[VH91] P. Van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.