Research Note

Draft (do not distribute)

Reconciling Finite Domains And Constrained Sorts

Denys Duchier

Draft of 23 November 1995

Contents

1 Introduction

Logic programming derives much of its appeal from the fact that it allows computation over incompletely specified objects where the missing parts are filled in as necessary as the computation proceeds. This is what terms containing logic variables are about. Constraint logic programming extends this approach and allows constraints to be imposed on the missing parts. The desired answer is thus incrementally approximated by constraining the shape and domain of the missing parts.

In Life, this view of logic programming as operating on approximations is at the very foundation of the language and is reflected in the notion of a ψ-term. A ψ-term is a *sorted extensible record* (see e.g. [2]). The sort hierarchy generalizes the distinction between bound and unbound variables: in Life, there are no variables, only more or less specialized ψ-terms; what used to be an unbound variable is now an instance of the top sort (with no features).

Constraint logic programming over finite domains also encourages the user to think of certain variables (namely finite domain variables) as incremental approximations of integer values. My purpose in the rest of the paper is to take a look at the correspondance between finite domains and sort hierarchies, to identify a technical difficulty with this view when constraints (intentional filters) are attached to sorts, and to propose a resolution of that problem which in turn suggests a generalization of the finite domain perspective.

2 Finite Domains And Sort Hierarchies

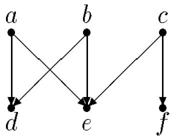
A finite domain variable X is an approximation of a value by the set of values which are consistent with all current constraints on X. For example, CLP(FD) [3] implements finite domains over small natural numbers and all constraints have the form X **in** r where r is a range expression.

A hierarchy of sorts is a richer structure than a set of ground values. Each sort may be regarded as an approximation of its subsorts. However, unlike a finite domain variable whose range can, in principle, be any subset of values, a sorted variable can only range over those distinguished subsets corresponding to the elements of the sort lattice.

Unless we have a complete lattice, in order to recover a flexibility of range similar to that of finite domain variables, it is necessary to introduce the notion of a *disjunctive* sort, i.e. a finite disjunction of elements of the sort lattice. Thus $\{s_1; \ldots; s_n\}$[1] approximates all the sorts in the union of their downsets.

Unifying sorted variables requires computing the GLB of their sorts. Efficient lattice operations can be supported using e.g. a bit vector encoding of sorts. As noted by Aït-Kaci *etal.* [1], disjunctive sorts can be naturally captured by such an encoding and require no modification of the GLB operation (typically bitwise-and).

Therefore, in principle, we can lift the idea of computing with finite domains to that of computing with disjunctive sorts.

Example: Consider the sort hierarchy: 

and consider the sorted variable $X : \{b; c\}$ whose disjunctive sort approximates the union of the downsets of b and c i.e. $\{b, c, d, e, f\}$. If we add the further constraint $X : a$, then the consistent domain of X shrinks down to $\{d; e\}$.

For the purpose of illustration, we will choose to represent sorts using the straightforward bit-vector encoding of their downset. The necessary computations can be carried out efficiently on this representation:

[1] This is Life syntax for the disjunction of sorts s_1 through s_n.

disjunction is bitwise-or and conjunction is bitwise-and.[2]

	a	b	c	d	e	f	
a	1	0	0	1	1	0	
b	0	1	0	1	1	0	
c	0	0	1	0	1	1	
$\{b;c\}$	0	1	1	1	1	1	
$a\&\{b;c\}$	0	0	0	1	1	0	$\equiv \{d;e\}$

3 Constrained Sorts

In a language such as Life, however, every sort s can be equipped with a set of constraints $c(X)$ which must be satisfied by every instance X of s. Without loss of generality, we can assume that there is exactly one constraint $c(X)$ per sort s.

Sort constraints are mainly used in two complementary capacities: first, to filter out certain refinements which do not satisfy a coherence requirement; second, as a means of adding sort specific information to a ψ-term (e.g. *all men have male gender*).

Let us write $X : \{s_1;\ldots;s_n\}$ for a variable X ranging over a disjunctive sort $\{s_1;\ldots;s_n\}$, and $c_i(X)$ for the constraint associated with sort s_i. It may well be that $c_i(X)$ is inconsistent with the current context, in which case we would like s_i to be removed from X's range, e.g. by adding the constraint $X : \neg s_i$.

In fact, all s_i may similarly be incompatible with the context, in which case we should like (indeed require) that the computation fail immediately.

Consider the earlier example: now suppose that e imposes the constraint $X.\ell = 1$ that X's feature ℓ must be 1, and that f imposes $X.\ell = 2$. Now, if $X.\ell$ is sofar undetermined, both refinements remain possible. If $X.\ell$ is known to be 1, then X is inconsistent with the constraint on f; therefore f must be removed from its range and we conclude that $X : e$. Finally, if $X.\ell$ is known to be, say, 3, then it is incompatible with both e and f; as a consequence, its range is reduced to the empty disjunction and the computation fails and backtracks.

In general, to determine that a constraint is inconsistent with the current context requires showing that none of its instances (specializations) can be derived. For this reason, in WILD Life, the decision was made to always enumerate disjunctive sorts: thus, having committed to an alternative s_i we can simply add $c_i(X)$ to the goal expression to enforce the satisfaction of its associated constraint. Unfortunately, this approach nullifies our attempt to use disjunctive sorts à la finite domains.

In this paper, I propose a way of reconciling constrained sorts with the finite domain view of disjunction. My approach is both semantically sound and lends itself to an efficient implementation that preserves incrementality.

4 A Calculus With Entailment

In [7] Gert Smolka describes a calculus for concurrent constraints with deep guards. It is parametrized by a constraint theory Δ and includes a conditional:

$$\textbf{if } E \textbf{ then } F \textbf{ else } G$$

or, more generally:

$$\textbf{if } \exists\bar{x}(E \textbf{ then } F) \textbf{ else } G$$

[2] Negation is bitwise-complement.

We are not concerned with the specifics of the calculus here, we merely wish to make use of the conditional construct and its semantics.

In particular, constraints can be propagated into the guard of a conditional:

$$\pi \wedge \textbf{if } E \textbf{ then } F \textbf{ else } G \equiv \pi \wedge \textbf{if } \pi \wedge E \textbf{ then } F \textbf{ else } G$$

The reduction rules for the conditional capture the notion of *entailment* and *disentailment*:

$$\begin{array}{ll} \textbf{if } \bot \wedge E & \textbf{then } F \textbf{ else } G \longrightarrow G \\ \textbf{if } \quad \top & \textbf{then } F \textbf{ else } G \longrightarrow F \end{array}$$

More precisely, for entailment:

$$\textbf{if } \exists\bar{x}(E \textbf{ then } F) \textbf{ else } G \longrightarrow \exists\bar{x}(E \wedge F) \qquad \text{when } \exists\bar{x}E \equiv \top$$

Given a variable X ranging over a disjunctive sort $\{s_1; \ldots; s_n\}$, we wish to filter out as soon as possible those s_i whose associated constraint $c_i(X)$ becomes inconsistent with the current context. We achieve this by posing constraints of the form: **if** $c_i(X)$ **then** $\top$ **else** $X : \neg s_i$ which simultaneously check for entailment or disentailment of $c_i(X)$. In case the constraint $c_i(X)$ becomes inconsistent with the store, we further impose that X cannot be a specialization of s_i — whether there is an efficient way of representing the complement of a sort entirely depends on the encoding method used and does not affect the present argument.

Unfortunately, if X suddenly becomes a specialization of s_i before $c_i(X)$ is entailed by the context, we must add $c_i(X)$ to the goal to *enforce* its satisfaction, thereby redoing much of the work already effected by incremental simplification of the conditional **if** $c_i(X)$ **then** $\top$ **else** $X : \neg s_i$.

Instead, we would rather somehow switch from passively checking for entailment or disentailment of $c_i(X)$ to actively searching for a resolution-based proof. To achieve this objective, I propose a small extension of Smolka's calculus for CCP where guards now have the form $E?x$:

$$\textbf{if } E?x \textbf{ then } F \textbf{ else } G$$

x can be interpreted as indicating whether E *must* be true. $?\top$ can be regarded as a kind of modal operator *must be true*, whereas $?\bot$ is an identity. We replace Smolka's reduction rules for the conditional by the ones below:

$$\begin{array}{lll} \textbf{if } (\bot \wedge E)?x \textbf{ then } F \textbf{ else } G & \longrightarrow & x = \bot \wedge G \\ \textbf{if } \top?x \textbf{ then } F \textbf{ else } G & \longrightarrow & F \\ \textbf{if } E?\top \textbf{ then } F \textbf{ else } G & \longrightarrow & E \wedge F \end{array}$$

Every sort constraint $c_i(X)$ can now be attached to a ψ-term X using two conditionals:

$$\begin{array}{ll} & \textbf{if } c_i(X)?t \textbf{ then } \top \textbf{ else } X : \neg s_i \\ \wedge & \textbf{if } X : s_i \textbf{ then } t = \top \textbf{ else } t = \bot \end{array}$$

This technique is related to the notion of *reified constraints* discussed e.g. by Henz and Würtz in [4].[3]

5 Constraint Propagation

A legitimate question is, when we consider a disjunctive sort, and in fact any sort at all (since individual sorts can be regarded as singleton disjunctions), should we post only the conditionals corresponding to the maximal sorts or should we just go ahead and post them for *all* subsorts as well.

Our view of disjunctive sorts as a generalization of finite domains suggests that this choice may be regarded as the distinction between *partial lookahead* and *full lookahead.*

[3] Thanks to Serge Le Huitouze for pointing out this connection to me.

Partial Lookahead: only propagates when the bounds are modified. In the case of disjunctive sorts, the bounds are the maximal sorts (maximal elements of the downset).

Full Lookahead: propagates for every modification of the domain. In the case of disjunctive sorts, this means the complete downset, i.e. all the subsorts. In general, full lookahead is obviously a great deal more expensive than partial lookahead.

6 Efficient Implementation

We presuppose a system which performs incremental checking of entailment and disentailment simultaneously, e.g. using *relative* or *situated simplification* [5, 6]. x in $E?x$ can be viewed as a status of *necessity* associated with the store in which E is being simplified. When this status is set to $\top$, all local bindings are promoted to the top level and all suspended guards in the store are promoted as top level goals.

References

[1] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. Technical report, MCC, Jul 1988.

[2] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of life. Technical Report PRL-RR-11, DEC PRL, May 1993.

[3] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*. To appear.

[4] Martin Henz and Jörg Würtz. Using oz for college time tabling. In *International Conference on Practice and Theory of Automated Timetabling*. DFKI, 1995.

[5] Andreas Podelski and Peter Van Roy. The beauty and the beast algorithm: Quasi-linear incremental tests of entailment and disentailment. In Maurice Bruynooghe, editor, *Proceedings of the international Symposium on Logic Programming (ILPS)*, pages 359–374. MIT Press, Nov 1994.

[6] Andreas Podelski and Gert Smolka. Situated simplification. In *Proceedings of CP95*, 1995.

[7] Gert Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, DFKI, February 1994.