Towards Coq Formalisation of {log} Set Constraints Resolution

Catherine Dubois[1], Sulyvan Weppe[2],
1. ENSIIE, lab. Samovar, CNRS, Évry, France
2. ENSIIE, Évry, France

Abstract. The language {log} is a Constraint Logic Programming language that natively supports finite sets and constraints such as (non) equality and (non) membership. The set constraints resolution process is mathematically formalised by Dovier et al in [5] using rewriting rules. In this paper we present a formalisation in the Coq proof assistant of the term and constraint algebra, the rewriting rules and check all the examples given in the reference paper by applying the rewriting rules manually with the help of some tailored tactics. The main problem we encountered is the non-determinism captured by the rewriting rules, which prevents us from automating their application in Coq. However the rules for non-membership and set checking are deterministic. So we propose a function that iteratively applies the latter rules. We prove its correctness with respect to the corresponding rewriting rules. This work is a first step of a larger project whose objective is to provide a formally verified resolution process for {log} set constraints resolution.

1 Introduction

The language {log}[1] is a freely available implementation of CLP(SET), recently extended to include binary relations and partial functions [4]. This language embodies the fundamental forms of set and a number of primitive operations for set management. It includes constraints for constructing and manipulating finite sets. In this paper we focus on the resolution of set constraints as it is detailed by Dovier et al in [5], considered in the following as the reference paper.

We contribute a formalisation within the Coq proof assistant [10] of the {log} resolution process of set constraints, or more precisely a first step towards this objective. Our motivation is to have a mechanized formal basis, in order to have a reference that could be used to study extensions, like the recent ones about partial functions and relations.

The resolution process extends the unification on first-order terms by adding specific set constraints e.g. (non) membership, (non) equality.

There are many formalisations of first-order unification in proof assistants, e.g. [8,9,2,1,7,6]. We can also mention a Coq proof of unification modulo associativity and commutativity with a neutral element embedded in the library Coccinelle [3] and also several proofs about nominal unification, e.g. [11].

[1] `http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html`

The work described in this paper is the first step of a larger project whose objective is to provide a formally verified resolution process for {log} set constraints resolution.

In this paper we present, in Section 2, the formalisation of the term and constraint algebra and the rewriting rules used in the set constraints resolution of {log} as exemplified in [5]. We go a step further by introducing some automation in the rewriting strategy. We propose to turn some of the rewriting rules into an operational process. It is described in Section 3 and we prove its termination and correctness. In Section 4, we conclude and present future work.

2 Coq Formalisation

In this section, we present first the formalisation of the term and constraint algebra and then the way we have formalised the rewriting rules used in the set constraints resolution of {log} as exemplified in [5]. Coq code is available at `http://www.ensiie.fr/~dubois/CoqSetlog`.

2.1 Terms and Constraints

A term is either a variable, the emptyset, a non-empty set or any non-set term built from a function symbol and a list of terms (let us call them ordinary terms). The type of terms, `term`, is represented in Coq as the following inductive datatype:

```
Inductive term: Set :=
| Var: variable → term
| SetC: term → term → term
| OTerm: symbol → list term → term
| EmptySet: term.
```

Ordinary terms are represented as varyadic terms. If necessary, we use a predicate for checking the well-formedness of such a term (stating that the length of the list of sub-terms is equal to the arity of the function symbol). The types of variables and symbols are any arbitrary types equipped with a decidable equality. Non empty sets are denoted by set terms of the form $\{a|t\}$, represented in Coq by `SetC a t`: a denotes an element of the set and t the set of the other elements. This notation stands for the set $\{a\} \cup t$. The function symbol $\{\_|\_\}$ used to construct sets is such that: (i) duplicates in a set do not matter and (ii) the order of elements is irrelevant. Both facts are taken into account by the resolution process.

The primitive constraints are equality (`Eq`), non-equality (`Neq`), membership (`Mem`), non-membership (`Nmem`) and set term constraints (`IsSet`). The type of primitive constraints is again defined as an inductive data-type. `FalseC` is added (wrt to the reference paper) to indicate that the resolution fails. A constraint is defined as a conjunction of primitive constraints, represented in Coq as a list of primitive constraints.

```
Inductive primitiveConstraint: Type :=
| Eq: term → term → primitiveConstraint
| Neq: term → term → primitiveConstraint
| Mem: term → term → primitiveConstraint
| Nmem: term → term → primitiveConstraint
| FalseC: primitiveConstraint
| IsSet: term → primitiveConstraint.

Definition constraint:=list primitiveConstraint.
```

Let pc be one of the primitive constraint of the constraint C. pc is in solved form if it has any of the following forms: (i) $X = t$, and neither t nor the rest of C contains X; (ii) $X \neq t$, and X does not occur in t; (iii) $t \notin X$, and X does not occur in t; (iv) $IsSet(X)$. A constraint C is in solved form if it is empty or all its components are in solved form.

We define also functions for checking occur-check, applying a substitution and some more helpers. We try to use as much as possible Coq notations to ease the reading and make our formalisation look like the paper presentation. For example the construct $\{t1|t2\}$ represented in Coq by `SetC t1 t2` is written in Coq `{{ t1 | t2 }}`. The constraint of equality $t1 = t2$ (resp. $t1 \neq t2$) is written `t1 == t2` (resp. `t1 /== t2`), $x \in t$ (resp. $x \notin t$) is written `x :s t` (resp. `x /:s t`).

2.2 Rewriting Procedures

In [5], the constraint solver is defined as a procedure named SAT_{SET} that nondeterministically transforms a constraint to either false, error, or a finite collection of constraints in solved form. This solver uses different rewriting procedures, one per kind of primitive constraints, to rewrite a set constraint to its equivalent solved form. Each rewriting procedure, made of different rules, models one step of rewriting. And each rule corresponds to a certain case of primitive constraint.

In our Coq formal development, we now formalise each single rewriting procedure as an inductively defined predicate. Each rule is translated into a clause of the predicate. We try to be very close to the original definitions.

In a constraint, once an equality $X = t$ where X does not occur in t, has been processed, it can be isolated because X has been eliminated from the rest of the constraint as it is usually done for first-order unification (e.g. in Color [2] and Coccinelle [3]). We follow the same approach and introduce the notion of *problem* as a pair whose first component contains solved equalities - called the solved part - and second component is a constraint to be rewritten - called the unsolved part. The type `problem` is defined in Listing 1.1. All Coq rewriting procedures share the type `problem → problem → Prop`, showing that a rewriting procedure rewrites a problem into another one.

At the beginning of the resolution, the solved part is empty. When the rewriting process is complete (that is no more rules can be applied), either the unsolved part contains `FalseC` meaning that a dead-end has been reached or the unsolved part provides us with a constraint in solved form.

All the Coq rules share the same format as exemplified in Listing 1.1: the first predicate specifies the position of the primitive constraint to be rewritten and its form, then optional conditions are stated and finally the new problem is specified. Usually in a rule (e.g. `stepMem2_2` in `stepMem` procedure) the targetted primitive constraint is removed and replaced in the unsolved part by one or some other primitive constraints. In some others, e.g. `stepMem1`, when a contradiction is found, the unsolved part is replaced by [`FalseC`], stopping the rewriting process.

```
Definition problem := constraint * constraint.

Inductive stepEq: problem → problem → Prop :=
...
|stepEq5: forall X t c1 c2 l1 l2 res,
     c2 =  l1 ++ [Var X == t] ++ l2 →
     ¬(occurCheck X t) →
     ((setTerm t)∨ ¬(isSetInC X c1) ∨ ¬(isSetInC X c2))→
     res = applySubst [(X,t)] (l1 ++ l2) →
     stepEq (c1, c2) ((Var X == t)::(applySubst [(X,t)] c1), res)
...

Inductive stepMem : problem→ problem→ Prop :=
|stepMem1: forall t c1 c2 l1 l2,
     c2 = l1 ++ [t :s EmptySet] ++ l2 →
     stepMem (c1, c2) (c1, [FalseC])
|stepMem2_1: forall r s t c1 c2 l1 l2 res,
     c2 = l1 ++ [r :s {{ s | t }}] ++ l2 → res = (r == s)::(l1++l2) →
     stepMem (c1, c2) (c1, res)
|stepMem2_2: forall r s t c1 c2 l1 l2 res,
     c2 = l1 ++ [r :s {{ s | t }}] ++ l2 → res = (r :s t)::(l1++l2) →
     stepMem (c1, c2) (c1, res)
|stepMem3: forall t X N c1 c2 l1 l2 res,
     c2 = l1 ++ [t :s (Var X)] ++ l2 →
     isFreshC N c1→ isFreshC N c2→ isFreshT N t → N<>X →
     res = (l1 ++ l2) ++ [Var X == {{ t | Var N}} ; IsSet (Var N)] →
     stepMem (c1, c2) (c1, res).
```

Listing 1.1. Coq encoding of some rewriting rules

These rewriting predicates are packed in one single, named `step`, specifying that a step in the resolution is achieved by one of the 5 previous predicates:

```
Inductive step : problem→ problem→ Prop:=
| step1 : forall pb pb', stepEq pb pb' → step pb pb'
| step2 : forall pb pb', stepMem pb pb' → step pb pb'
| step3 : forall pb pb', stepNeq pb pb' → step pb pb'
| step4 : forall pb pb', stepNmem pb pb' → step pb pb'
| step5 : forall pb pb', stepSC pb pb' → step pb pb'.
```

We define the transitive reflexive closure of each predicate, so that these closures allow us to achieve a complete transformation. The Coq standard library provides the predicate `clos_refl_trans` that defines the transitive reflexive closure of a binary relation:

```
Definition stepSCStar := clos_refl_trans  _ stepSC.
Definition stepStar := clos_refl_trans  _ step.
```

Using this formalisation, we can prove examples 2-4 of the reference paper with all their solutions. It is quite painful because we need to apply manually each rule. However the proof script is readable and allows us to follow precisely the different steps.

3 Towards More Automation

As said previously, the rewriting procedures are not deterministic, but actually some are. This is the case of `stepNmem` and `stepSC`. It is useful to define a functional version of these two ones, since the predicate versions we defined only allow us to make one step at once, whereas such a functional version would allow us to apply these steps iteratively, until we cannot anymore.

So we define the function `stepsSCheck` that iteratively applies the different rules of the rewriting predicate `stepSC`. This function implements a general recursive scheme: some cases do add some primitive constraints in the problem. Proof of termination in that case is not automatic in Coq. We easily prove the termination by introducing a dedicated measure on the constraint argument.

We prove the soundness of the function `stepsSCheck` with respect to the corresponding rewriting rules. Precisely, we prove, in Lemma `stepsSCheck_sound` below, that the result obtained by `stepsSCheck` is indeed in the reflexive transitive closure of the relation `stepSC`. However we do not use `stepSCStar` previously defined but an adaptation of it, `stepSCRd` which is defined as `stepSCStar` extended with a rule that allows us to pass over any constraint different from a `SC` constraint. We also prove the completeness of the function in Lemma `stepSC_complete`: if we can not rewrite a problem anymore with the `stepSC` rules, then `stepsSCheck` acts as the identity function. Again we use a variant of the rewriting relation `stepSC` to deal with `FalseC` that stops any computation in the function.

```
Lemma stepsSCheck_sound: forall c c1, stepSCRd (c1,c) (stepsSCheck (c1,c)).

Lemma stepsSCheck_complete: forall pb pb',
(forall pb', ¬ stepSCOnly (c1,l) pb') → stepsSCheck (c1,l) = (c1,l).
```

We can follow the same approach on `stepNmem`, resulting in a function correct with respect to the reflexive transitive closure of `stepNmen`. This is future work.

4 Conclusion

This paper presents the initial work done for formalising in Coq the set constraints resolution of {log}. We mainly define the term and constraint algebra and the different rewriting procedures, being as close as possible to the reference paper. All the examples presented in [5] are re-played in Coq, which brings

some relative confidence in our formalisation. The difficulties we encountered come from the fact that the rewriting procedures are not deterministic. So the definition of a resolution procedure cannot be done using functions (as done for first-order unification [2]). However for each deterministic rewriting relation, we can define a function that applies its rules until a solved form is achieved.

We propose several directions for future work. First, we want to formalise in Coq the main propositions and theorems of [5], such as termination and correctness of the resolution process. The second direction concerns the definition of a Coq function that implements the resolution process as it is proposed in [5] and its formal correctness proof. We have already achieved a first step, as explained in Section 3. The next step is a large one because it requires to implement backtracking in Coq, considered as implicit in the reference paper. An alternative to implementing backtracking could be to formalise an *all solutions* semantics, where the computed result represents the disjunction of the results of all the possible computations. This is another direction for future work.

Acknowledgements We thank M. Cristia and G. Rossi for the discussions which initiate this work. We thank G. Rossi for his help when we started the Coq formalisation. Thanks also to the anonymous reviewers for their suggestions.

References

1. A. B. Avelar, A. L. Galdino, F. L. C. de Moura, and M. Ayala-Rincón. First-order unification in the pvs proof assistant. *Logic Journal of the IGPL*, 22(5):758–789, 2014.
2. F. Blanqui and A. Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.
3. E. Contejean. Coccinelle, a Coq library for rewriting. In *Types*, Torino, Italy, 2008.
4. M. Cristiá, G. Rossi, and C. S. Frydman. Adding partial functions to constraint logic programming with sets. *TPLP*, 15(4-5):651–665, 2015.
5. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000.
6. S. Kothari and J. Caldwell. A machine checked model of idempotent MGU axioms for lists of equational constraints. In M. Fernández, editor, *Proc. 24th Int. Workshop on Unification, UNIF 2010, Edinburgh, United Kingdom, 14th July 2010.*, volume 42 of *EPTCS*, pages 24–38, 2010.
7. C. McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.
8. L. C. Paulson. Verifying the unification algorithm in lcf. *Sci. Comput. Program.*, 5(2):143–169, June 1985.
9. J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions. technical report 1795, 1992.
10. The Coq Development Team. *Coq, version 8.7.* Inria, Aug. 2017. `http://coq.inria.fr/`.
11. C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.