

# Prototyping completion with constraints using computational systems

Hélène Kirchner and Pierre-Etienne Moreau

CRIN-CNRS & INRIA-Lorraine  
BP 239  
54506 Vandœuvre-lès-Nancy Cedex  
E-mail: Helene.Kirchner@loria.fr

**Abstract.** We use computational systems to express a completion with constraints procedure that gives priority to simplifications. Computational systems are rewrite theories enriched by strategies. The implementation of completion in ELAN, an interpreter of computational systems, is especially convenient for experimenting with different simplification strategies, thanks to the powerful strategy language of ELAN.

## 1 Motivations

Completion procedures, as many computational processes, can be formulated as instances of a schema that consists of applying rewrite rules on formulas with some strategy, until getting specific normal forms. In this sense they can be understood as computational systems [1], i.e. rewrite theories in rewriting logic, enriched by a notion of strategy for selecting interesting derivations. Symbolic constraint solvers can also be described by computational systems that rewrite constraints to their solved forms. Putting together completion and constraint solving rules leads to a specification of completion with constraints using exclusively computational systems. It should be emphasized that each computation step, including for instance substitution or search for positions in a term, is specified by a computational system. This specification is actually executable in the system ELAN, an interpreter of computational systems. Checking this expressive power of computational systems was a first motivation for this work.

Using symbolic constraints in completion processes is an attractive idea but leads to non-trivial problems. For efficiency reasons, simplification is essential in completion but more difficult in the context of constrained equalities. As solutions are not computed, a constrained equality is simplifiable if all its instances are simplifiable. But this is not enough and redundancy of the simplified formula has to be checked. A second motivation for this work was to design a completion process with constraints that gives priority to simplifications, and to compare it with completion without constraints. In order to clarify the combination of weakening by propagation and simplification, we wanted to perform experimentations with different strategies for simplification and propagation.

This particular implementation of completion with constraints was not aimed at efficiency but rather at studying different strategies. Thanks to the powerful

strategy language of ELAN, it appears very easy to experiment various possibilities in a flexible and high-level way.

## 2 Completion with constraints

Full definitions and further references can be found in [2]. A constraint is a first-order formula built on a signature  $\Sigma$ . Elementary constraints are two constants  $T$  and  $F$ , and equations  $(t =^? t')$  with  $t, t'$  in the set of terms  $\mathcal{T}(\Sigma, \mathcal{X})$ . Non-elementary constraints are obtained by conjunction. The notation  $\sigma_c$  is used to denote the substitution which is the most general solution of the constraint  $c$ , unique up to renaming.

A *constrained equality*, denoted  $(l = r \parallel c)$ , is given by two terms  $l$  and  $r$  and a constraint  $c$ . When a constrained equality  $(l = r \parallel c)$  satisfies, for a given simplification ordering  $>$  total on ground terms,  $\sigma_c(l) > \sigma_c(r)$ , it is called a constrained rewrite rule and written  $(l \rightarrow r \parallel c)$ .

Completion with constraints is based on the superposition rule that deduces from two constrained rules  $(l_1 \rightarrow r_1 \parallel c_1)$  and  $(l_2 \rightarrow r_2 \parallel c_2)$  a new constrained equality  $(l_2[r_1]_\omega = r_2 \parallel c_3)$  where  $\omega$  is a non-variable position in  $l_2$ , provided  $c_3 = c_1 \wedge c_2 \wedge (l_2|_\omega =^? l_1)$  is satisfiable. Here the constraint  $(l_2|_\omega =^? l_1)$  is crucial to implement the basic strategy, since the part  $l_2[r_1]_\omega$  in the deduced formula is only made of basic positions. Writing this equation as a constraint prevents further inference steps to be applied in the substitutions solutions of this constraint.

While the superposition rule generates new constrained formulas, simplification is crucial to eliminate redundant equalities or rewrite rules. Informally an equality is redundant if it can be proved from smaller ones. Three simplifications can be defined for a given set  $R$  of constrained rewrite rules. The constrained equality  $(g = d \parallel c_1)$  is *simplified* by  $(l \rightarrow r \parallel c) \in R$ , into  $(u = v \parallel c_2)$ ,

- *With constrained simplification*: if there exist a non-variable position  $\omega$  in  $g$ , and a match  $\mu$  from  $\sigma_c(l)$  to  $\sigma_{c_1}(g|_\omega)$ . Then  $u = g[\mu(\sigma_c(r))]_\omega$ ,  $v = d$  and  $c_2 = c_1$ .
- *With instantiated constrained simplification*: if there exist a non-variable position  $\omega$  in  $\sigma_{c_1}(g)$ , and a match  $\mu$  from  $\sigma_c(l)$  to  $\sigma_{c_1}(g)|_\omega$ . Then  $u = \sigma_{c_1}(g)[\mu(\sigma_c(r))]_\omega$ ,  $v = \sigma_{c_1}(d)$  and  $c_2 = T$ .
- *With standard simplification*: if  $c = T$  and there exist a non-variable position  $\omega$  in  $g$ , and a match  $\mu$  from  $l$  to  $g|_\omega$ . Then  $u = g[\mu(r)]_\omega$ ,  $v = d$  and  $c_2 = c_1$ .

Constrained simplification does not simplify the constraint part and constrained equalities with a reducible constraint have to be eliminated with an additional simplification process. Instantiated constrained simplification replaces a constrained equality  $(g = d \parallel c_1)$  by an unconstrained one  $(u = v \parallel T)$ , which has the disadvantage to lose basic positions, but covers the case of reducible constraints. For example, consider a set of constrained rewrite rules that contains  $(x \rightarrow 0 \parallel x =^? i(0))$  and the constrained equality  $(y * 0 = 0 \parallel y =^? i(i(0)))$ . With constrained simplification, the rule cannot be used to simplify the equality due to the restriction to non-variable positions. With instantiated constrained simplification, the rule simplifies the equality into  $(i(0) * 0 = 0 \parallel T)$ .

The application of a simplification rule consists in the replacement of the simplifiable constrained equality by the result of the simplification steps. However, to ensure redundancy of the simplifiable equality, another condition has to be satisfied by constrained simplification. If an instance  $(l = r \parallel \sigma)$  is used to simplify an instance  $(g = d \parallel \sigma_1)$  of  $(g = d \parallel c_1)$ , each term of the co-domain  $\mathcal{Ran}(\sigma)$  of  $\sigma$  has to be included in a term of the co-domain of  $\sigma_1$ . Formally, given two sets of terms  $S$  and  $S'$ , let us define the relation  $S \sqsubseteq S'$  if any term in  $S$  is a subterm of a term in  $S'$ . Let us call the condition  $\mathcal{Ran}(\mu\sigma_c) \sqsubseteq \mathcal{Ran}(\sigma_{c_1})$  the *redundancy criterion*. It is proved in [2] that for the three defined simplifications  $(g = d \parallel c_1)$  is redundant if either  $c$  is equivalent to  $T$ , or  $\mathcal{Ran}(\mu\sigma_c) \sqsubseteq \mathcal{Ran}(\sigma_{c_1})$ .

To check that a constrained equality is redundant, two special situations may occur. If the rewrite rule has a constraint equivalent to  $T$ , redundancy is always satisfied. This means that unconstrained rewrite rules can be used for simplification without checking the redundancy criterion. On the other hand, if the constrained equality  $(g = d \parallel c_1)$  has a constraint equivalent to  $T$  the redundancy criterion can never be satisfied. Moreover unconstrained equalities are not the only ones for which the criterion does not apply. Let us consider for instance the constrained equality  $(k(f(h(y))) = k(y) \parallel y =^? b)$ . It is simplifiable using  $(f(x) \rightarrow x \parallel x =^? h(b))$  into  $(k(h(b)) = k(y) \parallel y =^? b)$ . But the redundancy criterion is not satisfied. Propagation is then needed to prove that such equalities are redundant. Brute force propagation simply generates from  $(l = r \parallel c)$ , the unconstrained formula  $(\sigma_c(l) = \sigma_c(r) \parallel T)$ . Indeed basic positions are then lost.

The problem is to find a practical tradeoff between simplification and propagation and to experiment with three approaches.

- A first possibility is to use propagation and standard simplification. Once the constraint has been propagated, the instantiated (unconstrained) rule can be used to simplify in a standard way any term in another formula. But if we propagate too often, benefit from constraints is lost. By instantiation, basic positions are forgotten and the completion with constraints is roughly equivalent to standard completion.
- A second possibility is to use constrained simplification and check redundancy of the simplified formula. Some redundancies may remain but can be eliminated as late as possible using propagation. Note that the completion process can diverge if propagation is not applied.
- A third possibility is to use instead instantiated constrained simplification and standard simplification whenever possible.

### 3 Implementation in ELAN

In this section, we give an overview of how completion with constraints is implemented in ELAN using computational systems [1]. A computational system is given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and a strategy to guide application of rewrite rules. Formally, this is the combination of a rewrite theory in rewriting logic, together with a notion of strategy to efficiently compute with given rewrite rules.

The underlying framework is first-order, since formulas, sets of formulas and proofs are considered as first-order terms. Computation is exactly application of rewrite rules on a term and a strategy describes the intended set of computations, or equivalently in rewriting logic, a subset of proof terms. A computational system may be generic in the sense that it is parameterized by a class of signatures and axioms. In this case to each program, i.e. each pair of a signature and a set of axioms, corresponds by instantiation, a computational system.

For the specification of completion with constraints, the syntax, the rewrite rules and the strategies are described in different modules that are just partially developed here. A module has a name and a list of parameters. It can import other modules and contains sort declarations, operator declarations, rules and strategies.

The main data structure is a `compute_state` described in a module parameterized by a set of variables `vars`, a set of function symbols `fss`, ordered by a precedence relation `prec`, a set of objects `X` (that will be constrained equalities). A `compute_state` is implemented using six lists.

```

module compute_state[vars,fss,prec,X]
import list[X]
sort compute_state ;
op
  @*@@*@@*@@ : (list[X] list[X] list[X] list[X] list[X] list[X])
                compute_state;
endop ...
end of module

```

The top-level computational system imports this data structure and contains the rules for completion with constraints. It also imports other modules describing constrained equalities, constraints, unification and matching, rewriting and a module defining the used simplification strategy. Rules, adapted from those describing the ANS-completion in the ORME system, are applied to a `compute_state` until a specific form is obtained where only the first component *A* is non-empty. Variables used in these rules are declared with their sorts. These rules exemplify the syntax of ELAN. In the `where` part, local variables are instantiated by the result of evaluating sub-strategies, for instance `simplify_ANCT` in the first rule, calling itself simplification by the rule `s` on lists *A,N,C,T*.

In ELAN, a successful application of a rewrite rule on a given term is processed in the expected way. The left-hand side of the rule has to match a subterm of the given term; this causes instantiation of variables occurring in the left-hand side. Then the local assignments and conditions are evaluated, in reverse order, so that the evaluation starts with the last one and stops with the first one. The conditions have to be evaluated to a constant *true* and the local assignments must instantiate all remaining variables. At the end, the original subterm is replaced by the instantiated right-hand side of the rule. In the `where` assignments, the terms are reduced with respect to strategies and the results are then assigned to the variables on the left-hand sides of the `:=` assignment signs.

```

module cans_completion[vars,fss,prec,syst]

```

```

import  simplify_basique[vars,fss,prec]
        eq_constraint[vars,fss,prec]
        unif_match[vars,fss]
        constraint[vars,fss]
        rewrite[vars,fss,prec]
        compute_state[vars,fss,prec,eq_constraint]
rules for compute_state
declare
    A,N,C,T,S,E      : list[eq_constraint];
    n,c,t,s,e        : eq_constraint;
    ln,lt,le         : list[eq_constraint];
    cs1,cs2,cs3      : compute_state;
bodies
  [Rule1] A * N * C * T * s.nil * E          => S_to_T(cs1)
          where cs1:=(simplify_ANCT) A * N * C * T * s.nil * E
end
  [Rule2] A * N * C * T * nil * e.le => cs3
          where cs3:=(orient_E) cs2
          where cs2:=(delete_E) cs1
          where cs1:=(clean_E) A * N * C * T * nil * e.le
end
  [Rule3] A * n.ln * c.nil * T * nil * nil   => cs1
          where cs1:=(deduce_NC) A * n.ln * c.nil * T * nil * nil
end
  [Rule4] A * nil * c.nil * T * nil * nil     => AC_to_N(cs1)
          where cs1:=(internal_deduce) A * nil * c.nil * T * nil * nil
end
  [Rule5] A * N * nil * t.lt * nil * nil      => cs1
          where cs1:=(choice_C) A * N * nil * t.lt * nil * nil
end

```

The general strategy called **completion** consists in applying deterministically these five rules in the given order until no rule applies anymore.

```

strategy completion
  while
    dont care choose(Rule1 Rule2 Rule3 Rule4 Rule5)
  endwhile
end of strategy

```

Strategy definitions in ELAN are expressed in a strategy language, that includes the rule names, a concatenation operator (omitted in the syntax), iterator operators **while** and **repeat** and choice operators. Note that rule names are always encapsulated in a choice operator, because a rule name represents in general a set of rules. The **dont know choose** operator is a choice operator that, given  $n$  arguments, provides one of them and if its evaluation fails, provides another one. The **dont care choose** operator, contrary to **dont know choose**, selects only one of sub-strategies giving a non-empty set of results. To increase the expressive power of expressions built on this vocabulary, there is the possibility to express application of a strategy on subterms, thanks to the **where** mechanism

in rewrite rules. Moreover strategies can be named and these names can be used in the definition of other strategies.

To complete a system of equalities, the user has to provide it in the following form.

```
specification p
Vars    x  y  z ;
Ops     f:2 e:0 i:1 ;      % operators with their arity
Prec    e:1 f:2 i:3 ;      % precedence e<f<i
System
    f(f(x,y),z) = f(x,f(y,z)) .
    f(e,x)=x .
    f(x,i(x))=e .
    nil
end of specification
```

In order to make experiments, several strategies for simplification and propagation have been tested. For instance, the following strategy `simplify` applies as long as possible, standard simplification, then propagation.

```
strategy simplify
while
    dont care choose(st_simplify propagate)
endwhile
end of strategy
```

Other strategies can be defined in a similar way to experiment with constrained simplification and instantiated constrained simplification. We also compared completion with constraints (`cans_completion`) with standard completion on several examples, in order to check the gain of using constraints. This gain is measured by the number of applications of the superposition rule. For instance, on the classical example of groups, `cans_completion` applied to the previous specification computes the result by applying 128 superposition steps instead of 219 in the case of a standard completion without constraints.

## How to get the system:

ELAN is available by ftp at <ftp://ftp.loria.fr/pub/loria/protheo/software/Elan>.

## References

1. Claude Kirchner, Hélène Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. MIT press, 1995.
2. H. Kirchner and P-E. Moreau. Prototyping completion with constraints using computational systems. Technical Report 94-R-201, CRIN, 1994.