Completion Procedures as Transition Rules + Control

Pierre LESCANNE*
Centre de Recherche en Informatique de Nancy
LORIA
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy, France
email: lescanne@poincare.crin.fr

Abstract

A description of the completion of a set of identities by a set of inference rules has allowed recent progresses in proving its completeness. But there existed no attempt to use this description in an actual implementation. This paper shows that this is feasible using a functional programming language namely *CAML*. The implementation uses a toolkit, a set of transition rules and a short procedure for describing the control. A major role is played by the data structure on which both the transition rules and the control operate. Three versions of the classical Knuth-Bendix completion and two versions of the unfailing completion are proposed.

1 Completion procedures as sets of transitions rules

The interest of rewriting techniques in programming, algebraic and computer algebra specifications is well-known as is its ability to provide proof environments essentially based on completion procedures [FG84,GG88,KS83,Fag84,Les83]. In this introduction, I suppose the reader is familiar with this concept. Indeed my goal is not to present it, but to study how methods developed essentially with a theoretical purpose, namely proving completeness can be used to present simple short and understandable programs. This paper can also be seen as a set of exercises on the use of a functional language to program high level procedures and as a bridge between theory and practice. Readers who want to get more introductory informations are invited to look at Appendix A or to Dershowitz survey *Completion and its Applications* [Der87]. The completion procedure is a method used in equational logic to built from a set of identities an equivalent canonical set of rewrite rules i.e., a confluent, noetherian and interreduced set of rules used to compute normal forms. If one tracks the history of the presentation of this procedure, one can notice different methods of description. In their seminal paper [KB70] Knuth and Bendix describe essentially the procedure in natural language (see Appendix B), in [Hue80] Huet uses a style similar to Knuth's book, *The Art of Computer Programming*, in [Hue81] he uses a program structured by while loops, in [Kir84] H. Kirchner uses a recursive procedure and in [For84] Forgaard proposes an organization of the procedure around tasks to be performed. In the following, a

*The research was sponsored by PRC "programmation avancée et outils de l'intelligence artificielle", CNRS and INRIA

completion will be seen as a set of *inference rules* or more precisely a set of *transition rules* acting on a data structure. The idea of using *inference rules* when dealing with completion is not new and leads to the beautiful proofs of completeness proposed by Bachmair and Dershowitz [Bac87,BDH86,BD87] and their followers [GKK88,Gan87]. The *completeness* is the ability of the procedure to eventually generate a proof by normalization or a rewrite proof for every equational theorem. In this paper, I want to show how this description leads to actual, nice and elegant programs when used as a programming method and I illustrate that by an actual *CAML* implementation [FOR87b]. Appendix C gives the basic notions that are useful to understand the programs. Actually the inference rules one considers in completion are specific in the sense that they transform a t-uple of objects into a t-uple of objects with the same structure. This is why I refer to them as *transition rules.* Thus the basic components of such a procedure are four,

- **a data structure** on which the transition rules operate, sometimes called the universe,
- **a set of transition rules,** that are the basic operations on the data structure,
- **a control,** that is a description of the way the transition rules are invoked[1],
- **a toolkit** that is shared by several completion procedures.

When one wants to describe a specific completion procedure, usually one uses the following method. First one chooses the data structure, then one chooses transition rules and often at the same time the control. The toolkit is something that remains from one procedure to the other in many cases, it was partly borrowed from the "*CAML* Anthology" [FOR87c] as a natural attempt to reuse pieces of codes already debugged and tested. As we will see the control is typically data driven and can be easily expressed by rewrite rules. In the following, the influence of these choices on the efficiency of the procedure will be illustrated through three refinements of the Knuth-Bendix completion procedures and a two unfailing completions. Indeed, we will see how, starting from a naive implementation of the completion, improvements can be obtained by changing the data structure and consequently the transition rules and the control. These ideas are implemented in my program *ORME.*

2 The N-completion

In this section, I give a naive implementation of the completion, called the *N-completion,* where N stands for naive. It is already an improvement of the set of rules of Appendix A in order to take the computation of critical pairs into account. Its control part is fully given in Figure 1 and its data structure has three components, namely

- **E** is a set of identities, either critical pairs or given identities,
- **T** is a set of rules, the non marked rules in Huet's terminology [Hue81],
- **R** is a set of rules whose critical pairs have been computed, the marked rules.

In the procedure, *ordering* is a parameter which is a relation used to orient the identities into rules, by the way it is also a parameter of *Orientation.* There are three kinds of transition

[1] To give a gastronomic comparison [Ore83], the control is the recipe.

```
let rec N_Completion ordering (R,T,E) = let COMP = N_Completion ordering in
match (T,E) with
    [],[] ->  (R,[],[]) (* success *)

| (_::_),[] -> COMP (repeat_list [Simpl_left_T_by_T;Simpl_left_T_by_R;
                                 Simpl_left_R_by_T;Simpl_left_R_by_R;
                                 Simpl_right_T;Simpl_right_R]
                    (Deduction(R,T,E)))

| _,(_::_) -> let (R',T',E') = repeat_list[Remove_trivial_E;Simpl_E](R,T,E) in
              (match E' with
                  [] -> COMP(R',T',E')
                | (_::_) -> COMP(Orientation ordering (R',T',E')
                                ? failwith "non orientable equation"));;
```

Figure 1: *The N-completion*

rules, their names are taken according to Dershowitz [Der87] (see also Appendix A). *Deduction* computes critical pairs, in this case it computes critical pairs between one rule in T, usually the smallest one to be more efficient, and all the rules in R. *Orientation* chooses an identity that can be oriented by an *ordering* and produces a rule, if no identity is orientable, it fails. This requires an reduction ordering, currently *ORME* contains an ordering based on polynomial interpretations [BL87b], implementing other orderings would not be too difficult since the *CAML* Anthology [FOR87c] contains the recursive path ordering and a *CAML* implementation of the transformation ordering also exists [BL87a,Gal88]. *Remove_trivial_E* removes from E a trivial identity. The rules *Simpl_left_T_by_T*, *Simpl_left_T_by_R* etc. simplify terms in the rules or the identities. *repeat_list* repeats the application of of a list of inference rules until they all fail. The control given in Figure 1 has essentially three steps, namely *success,* when T and E are empty, *computing critical pairs* after simplification of the rules, when E is empty, and *orienting* an identity into a rule after simplification of the identities, when E is not empty. In the orientation part it could happen that by simplification all the identities disappear, in this case one does nothing, that is just translated by a recursive call to COMP with the same parameters. The recursive calls mean that one restarts the process. The completion terminates with success when E and T are empty. The system works as a machine where the identities enter E and proceeds through T and R. Its description is therefore really similar to this of an automaton.

3 The S-completion

Another name for rewrite systems is sometime *simplifying systems* and the theory of rewrite systems is a *theory of simplification*, that could be applied to many fields other than computer algebra or software specification. Therefore the main aim of orienting identities is to use them to simplify whenever it is possible. But as noted by Hsiang and Mzali [HM88], the *N-completion* makes a bad use of simplification. Indeed a rule is not used for simplification as soon as it has been generated. Thus in a better implementation, when an identity is oriented into a rule it enters a set S where it is used to simplify all the other identities and

```
let rec S_Completion ordering (R,T,S,E) = let COMP = S_Completion ordering in
match (T,S,E) with
    [],[],[] ->  (R,[],[],[]) (* success *)

| _,(_::_),_  -> (COMP (R',T'@S',[],E')
                    where R',T',S',E' =
                        repeat_list [Simpl_left_T_by_S;Simpl_right_T_by_S;
                                     Simpl_left_R_by_S;Simpl_right_R_by_S] (R,T,S,E))
| (_::_),[],[] -> COMP (Deduction (R,T,S,E))
                                                                          10
| _,[],(_::_) -> let (R',T',S',E') = repeat_list[Remove_trivial_E;Simpl_E](R,T,S,E)
                 in (match E' with
                       [] -> COMP(R',T',S',E')
                     | (_::_) -> COMP (Orientation ordering (R',T',S',E')
                                    ? failwith "non orientable equation"));;
```

Figure 2: *The S-completion*

rules. In the *S-completion*, the data structure is made of four components,

- **E** like in the *N-completion*,
- **S** a set of oriented identities or rules that are used to simplify others identities or rules and that I call the *simplifiers*, during the completion S contains zero or one rule,
- **T** a set of rules already used for simplifying, but whose critical pairs are not yet computed,
- **R** like in the *N-completion*.

The only difference with the *N-completion* is the set S through which a rule has to go, before entering T. The step of simplification is clearly distinguished from the three others. It is performed when S is not empty. The completion process ends when there is no more identity or rule in E, S and T.

4 The ANS-completion

The *S-completion* can still be improved since it computes at the same time the critical pairs between all the rules in R and one rule in T. It should be better to compute the critical pairs between one rule in R and one rule in T at a time. As previously, S contain the simplifiers. In addition, a set C is created to contain one rule extracted from T with which critical pairs with rules of R are computed. To keep track of the rules whose critical pairs are computed with the rule in C, R is split into two sets A (for already computed) and N (for not yet computed). Thus the data structure contains,

- **E** like in the *S-completion*,
- **S** like in the *S-completion*,

- **T** is a set of rules coming from S and waiting to enter C,
- **C** is a set that contains one or zero rule and whose critical pairs are computed with one in N,
- **N** is the part of R whose critical pairs have not been computed with C but whose critical pairs with $A \cup N$ have been computed,
- **A** is a set whose critical pairs with $A \cup N \cup C$ have been computed.

The transition rules are adapted to work with this new data structure and three new rules are introduced. *Deduction* computes the critical pairs between the smallest rule in N and the rule in C. *Internal_Deduction* computes the critical pairs obtained by superposing the rule(s) in C on itself (themselves). *A_C2N* moves the rules in A and C into N to start a new "loop" of computation of critical pairs, according to the emptyness of the components of the data structure. The procedure has now clearly six parts, namely *success, simplification, orientation, deduction, internal deduction* and *beginning of a new loop* of computation of critical pairs. Typically this cannot be easily structured by a while loop because at each time the iteration on the computations of the critical pairs can be interrupted by a simplification. A data driven control is then much better (Figure 3).

5 The unfailing completion

The previous method may fail because at certain time no rule can be oriented, this is for instance the case if $(x * y = y * x) \in E$. A method called either *unfailing completion* or *unfailing Knuth-Bendix* or *UKB* has been proposed by Hsiang and Rusinowitch [HR87] and Bachmair, Dershowitz and Hsiang [BDH86] and is complete for proving equational theorems in equational theories. The idea is to refute the equality to be proved, thus variables become Skolem constants, terms become ground terms and the equality becomes a disequality i.e., a negation of an equality. One does not orient the identities. But because one works with an ordering total on ground terms, one knows that for any pair of ground terms one can tell which one is simpler and therefore one can tell whether a term that matches a side of an identity can be transformed in the other side in a decreasing way. In some sense, these new pairs of terms are not rules but "abstract" sets of rules on ground terms. To precise the difference with identities and rules, let me propose the word *likeness* for them. The aim of unfailing completion is then to make confluent the rewrite relation on ground terms defined by likenesses. Such a relation which is confluent on ground terms is called *ground confluent*. Although the likenesses are not oriented, one tries however to save generation of too many critical pairs by not keeping those of the form $\langle s, t \rangle$ obtained from a superposition u by $s \longmapsto u \longmapsto t$ if either $u < s$ or $u < t$, because this kind of equality will never be used for rewriting ground terms. This is what makes this procedure different from classical paramodulation. As a refutation procedure, at each step, an attempt to refute the negation of the disequality is performed. The data structure of this naive unfailing completion is a follow

- **E** is again the set of identities,
- **C** is a set that contains one or zero likenesse and whose critical pairs are computed with one in N,

```
let rec ANS_Completion ordering (A,N,C,T,S,E) =
    let COMP = ANS_Completion ordering in

match (N,C,T,S,E) with
   _,[],[],[],[] -> (A,N,C,T,S,E) (* success *)

 | _,_,_,(_::_),_ -> (COMP (A',N',C',T'@S',[],E')
                          where A',N',C',T',S',E' =
                               repeat_list [Simpl_left_A_by_S;Simpl_right_A_by_S;
                                            Simpl_left_N_by_S;Simpl_right_N_by_S;
                                            Simpl_left_C_by_S;Simpl_right_C_by_S;
                                            Simpl_left_T_by_S;Simpl_right_T_by_S]
                                 (A,N,C,T,S,E))

 | _,_,_,[],(_::_) -> let  A',N',C',T',S',E' = repeat_list[Remove_trivial_E;
                                                           Simpl_E](A,N,C,T,S,E) in
                      (match E' with
                          [] ->  COMP(A',N',C',T',S',E')
                        | (_::_) ->  COMP(Orientation ordering (A',N',C',T',S',E')
                                     ? failwith "non orientable equation"))

 | (_::_),[_],_,[],[] -> COMP (Deduction(A,N,C,T,S,E))

 | [],[_],_,[],[] -> COMP (A_C2N crit (Internal_Deduction (A,N,C,T,S,E)))

 | _,[],(_::_),[],[] -> (COMP([],A@N,[r],T',[],[]) where r,T' = least Size T);;
```

Figure 3: *The ANS-completion*

```
let rec Unfailing_Completion ordering (e,A,N,C,E) =
        let COMP = Unfailing_Completion ordering in
let e' = Gnormalize  ordering  (A @ N @ C @ E) e in
if matches <<x ~ x>> e'
   then (e',A,N,C,E)  (* refutation *)
   else match (N,C,E) with
      _,[],[] ->    (e',A,N,C,E) (* end of the completion *)

    | _,[],(_::_) -> let (A',N',C',E') = Clean_E (Simpl_E ordering
                                                  (Simpl_N ordering (A,N,C,E))) 10
        in (match E' with
          [] -> COMP (e',A',N',C',E')
        | eq :: E'' -> COMP(e', F_Subsumption ordering ([], A' @ N', [eq], E'')))

    | (_::_),(_::_),_ -> COMP(e', (Deduction ordering (A,N,C,E)))

    | [],[_],_ ->  COMP(e', (A_C2N (Internal_Deduction ordering (A,N,C,E))));;
```

Figure 4: *The unfailing completion*

- **N** is a set f likeness whose critical pairs have not been computed with C but whose critical pairs with $A \cup N$ have been computed,
- **A** is a set whose critical pairs with $A \cup N \cup C$ have been computed.

It should be noticed that the idea of computing the critical pairs between only two pairs at a time is used, but not the idea of putting a high priority to simplification and since there is no simplification the identities enter directly C from E. With the disequality to be refuted, the procedure has five parameters. The last four ones remind the data structure of the classical completion. Obviously, there is no *Orientation* transition rule, but there are *Deduction* and *Internal_Deduction* as previously. << x ~ x >> is an external notation for the disequality whose both sides are x. There are four steps in this procedure, either success or, simplification, or deduction or, the beginning of a new "loop" of computation of critical pairs. If one runs this algorithm on examples, one quickly realizes that many generated identities are instances of existing identities or obtained by inserting in a same context sides of instances of identities and therefore do not carry new information. Rules *Subsumption* or *F_Subsumption* remove these useless identities. *Subsumption* filters the identities that matches another one and *F_Subsumption* tries to remove identities of the form $C[s] = C[t]$, where $C[\ ]$ is a context, when the identity $s = t$ already exists. *Gnormalize* takes a ground term and returns its normal form using identities. However, when rewriting with identities as in refutation care must be taken with variables that can be introduced. The usual solution is to instantiate them by a new *least constant*.

6 An improved unfailing completion: the ER-completion

The previous unfailing completion has the advantage of being short and easy to understand. However its main drawback is that it makes no difference between non orientable identities

and rules. This can be fixed by refining considerably the data structure, using ideas from the *ANS-completion*. The new data structure contains eleven components. They are obtained by splitting the corresponding components of the *ANS-completion* into two parts, a E-part and a R-part.

- **E** is not changed and is again the set of identities,
- **RS** the simplifiers obtained from rules E by orientation,
- **ES** the identities from E that cannot be oriented, they are used in *F_Subsumption*,
- **RT** the R-part of T,
- **ET** the E-part of T,
- **RC** the R-part of C,
- **EC** the E-part of C,
- **RN** the R-part of N,
- **EN** the E-part of N,
- **RA** the R-part of A,
- **EA** the E-part of A,

The transition rules are changed accordingly and one gets a procedure I call the *ER-unfailing completion* (see Figure 5) which performs as a classical completion if all the identities can be oriented. The fact that everything which is orientable is actually oriented is a major improvement for the efficiency of the procedure. The structure of the completion procedure gets now more complex and requires studies on how to make it more modular.

7 Conclusion

The main idea of the approach presented here is to decompose the algorithm into basic actions and to describe some kind of abstract machine where these actions as the instructions. This may remind either Forgaard's description of *REVE* based on tasks [For84], or *ERIL* [Dic85] where users have access to the basic operations or Huet's first description [Hue80]. The rigorous and formal approach of this paper gives precision and concision and leads to a better understanding of the program and therefore to a better confidence. Since one is closer to the proof of completeness there are more chance that the implementation is both correct and complete. Another important aspect of this approach is that modifications and improvements are easily done. Basically this level of programming allows to study very high level optimizations [Ben82] and when an efficient procedure is discovered, a low level implementation can be foreseen. Here I made many implementation choices that still can be discussed, but since they are rather explicit this discussion is easy and changes can be quickly made. However, as well illustrated by the *ER_completion* compared with the *unfailing completion*, it should also be noticed that the complexity of the completion procedures described by transition rules increases exponentially with the size of the number of components of the data structure, which implies that some kind of modularity has to be found.

```
let rec Unfailing_Completion ordering (e,((EA,RA,EN,RN,EC,RC,ET,RT,ES,RS,E) as STATE)) =
        let COMP = Unfailing_Completion ordering and ord = ordering in
let e' = Gnormalize  ordering (EA @ EN @ EC @ ET @ ES @ E) e''
 where e'' = (normalize (RA @ RN @ RC @ RS @ RT) e) in
if matches <<x ~ x> e'
   then  (print_state "REFUTATION" (e',STATE);(e',STATE))  (* refutation *)
   else match (EN,RN,EC,RC,ET,RT,ES,RS,E) with
      _,_,[],[],[],[],[],[],[] ->  (e',STATE) (* end of the completion *)

    | _,_,_,_,_,_,_,(_::_),_ -> (COMP(e',EA',RA',EN',RN',EC',RC',ET',RT'@RS',ES',[],E')
        where (EA',RA',EN',RN',EC',RC',ET',RT',ES',RS',E') = Simp_by_RS (STATE))

    | _,_,_,_,_,_,_,(_::_),[],_ -> (COMP(e,EA',RA',EN',RN',EC',RC',ET'@ES',RT',[],RS',E')
        where (EA',RA',EN',RN',EC',RC',ET',RT',ES',RS',E') =
                                        F_Subsume_by_ES ord (STATE))

    | _,_,_,_,_,_,[],[],(_::_) -> let ((EA',RA',EN',RN',EC',RC',ET',RT',ES',RS', E')
                                                          as STATE') =
          Clean_E ord (STATE) in (match E' with
                      [] -> COMP (e',STATE')                                    20
                    | (_::_) -> COMP(e',Orientation ord (STATE')))

    | _,(_::_),[],[],_,_,[],[],[] -> COMP(e', RN_RC_Deduction (STATE))

    | (_::_),_,[],[],_,_,[],[],[] -> COMP(e', EN_RC_Deduction ord (STATE))

    | _,(_::_),[],[],_,_,[],[],[] -> COMP(e', RN_EC_Deduction ord (STATE))

    | (_::_),_,[],[],_,_,[],[],[] -> COMP(e', EN_EC_Deduction ord (STATE))
                                                                              30
    | [],[],[],[],_,_,[],[],[] -> COMP(e', A_C2N (RC_Internal_Deduction(STATE)))

    | [],[],[],[],_,_,[],[],[] -> COMP(e', A_C2N (EC_Internal_Deduction ord (STATE)))

    | _,_,[],[],(_::_),[],[],[],[] -> (COMP(e',[],[],EA@EN,RA@RN,[e],[],ET',[],[],[],[])
                                        where e,ET' = (least Size ET))
    | _,_,[],[],[],(_::_),[],[],[] -> (COMP(e',[],[],EA@EN,RA@RN,[],[r],[],RT',[],[],[])
                                        where r,RT' = (least Size RT))
    | _,_,[],[],(_::_),(_::_),[],[],[] ->
       let r,RT' = least Size RT and e,ET' = least Size ET in
       if Size r <= Size e                                                      41
       then COMP(e',F_Subsume_by_ES ord ([],[],EA@EN,RA@RN,[],[r],ET,RT',[],[],[]))
       else COMP(e',F_Subsume_by_ES ord ([],[],EA@EN,RA@RN,[e],[],ET',RT.[],[],[]))
```

Figure 5: *The ER unfailing completion*

Another interesting aspect of the programming by transition rules is that simple snapshots exist, therefore the process can easily be stopped after each rule and restarted on this state. Thus backtracking on the choice of the orderings as implemented in *REVE* or any kind of backtracking to insure fairness [DMT88], backups, breakpoints or integration of an already completed rewrite system in another equational theory can be easily handled.

But this approach does not address low level controls, for instance refinements that computes one critical pair at a time. This indeed requires a level of granularity in the actions that cannot be handled by the current form of the data structure. Attempts to fully formalize all the tasks, including substitutions and unifications should answer this question [GS88,HJ88].

All the procedures presented in this paper are a part of *ORME*, a set of *CAML* procedures that were run for completing a set of examples. Both the programs and the examples can be obtained from the author upon request.

I would like to thank Leo Bachmair, Françoise Bellegarde, Jieh Hsiang, Jean-Pierre Jouannaud, Jean-Luc Remy, Michael Rusinowitch and the EURECA group at CRIN who provided me with stimulating discussions, Gérad Huet who gave me access to the *CAML Anthology* and Alain Laville for wise advices on how to use *CAML*.

References

[Bac87] L. Bachmair. *Proof methods for equational theories.* PhD thesis, University of Illinois, Urbana-Champaign, 1987.

[BD87] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. In *Proceedings Second Conference on Rewriting Techniques and Applications*, Springer Verlag, Bordeaux (France), May 1987.

[BDH86] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proc. Symp. Logic in Computer Science*, pages 346–357, Boston (Massachusetts USA), 1986.

[Ben82] J. L. Bentley. *Writing Efficient Programs.* Prentice Hall, 1982.

[BL87a] F. Bellegarde and P. Lescanne. Transformation orderings. In *12th Coll. on Trees in Algebra and Programming, TAPSOFT*, pages 69–80, Springer Verlag, 1987.

[BL87b] A. BenCherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–160, October 1987.

[Der87] N. Dershowitz. Completion and its applications. In *Proc. Colloquium on Resolution of Equations in Algebraic Structures*, MCC, 3500 West Balconies Center Drive, Austin, Texas 78759-6509, May 4-6 1987.

[Dic85] A.J.J. Dick. ERIL equational reasoning: an interactive laboratory. In B. Buchberger, editor, *Proceedings of the EUROCAL Conference*, Springer-Verlag, Linz (Austria), 1985.

[DMT88] N. Dershowitz, L. Marcus, and A. Tarlecki. Existence, uniqueness and construction of rewrite systems. *SIAM J. Comput.*, 17(4):629–639, August 1988.

[Fag84] F. Fages. *Le système KB. Manuel de référence, présentation et bibliographie, mise en œuvre.* Technical Report, Greco de Programmation, Bordeaux, 1984.

[FG84] R. Forgaard and J. Guttag. *REVE: A term rewriting system generator with failure-resistant Knuth-Bendix.* Technical Report, MIT-LCS, 1984.

[For84] R. Forgaard. *A program for generating and analyzing term rewriting systems.* Technical Report 343, Laboratory for Computer Science, Massachusetts Institute of Technology, 1984. Master's Thesis.

[FOR87a] Projet FORMEL. *The CAML Primer.* Technical Report, INRIA LIENS, 1987.

[FOR87b] Projet FORMEL. *CAML: the reference Manuel.* Technical Report, INRIA-ENS, March 1987.

[FOR87c] Projet FORMEL. *The CAML Anthology.* July 1987. Internal Document.

[Gal88] B. Galabertier. *Implémentation de l'ordre de terminaison par transformation.* Technical Report, CRIN, Septembre 1988.

[Gan87] H. Ganzinger. A completion procedure for conditional equations. In *Proc. 1st International Workshop on Conditional Term Rewriting Systems*, pages 62–83, Springer-Verlag, 1987. Extended version to appear in Journal of Symbolic Computation.

[GG88] S. Garland and J. Guttag. *An Overview of LP, The Larch Prover.* Technical Report, MIT, 1988.

[GKK88] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. In M. Dauchet and M. Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, pages 165–184, Springer-Verlag, Nancy (France), 1988.

[GS88] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Journal of Theorical Computer Science*, 1988.

[HJ88] J. Hsiang and J-P. Jouannaud. General e-unification revisited. In *Proceedings of 2nd Workshop on Unification*, 1988.

[HM88] J. Hsiang and J. Mzali. *Algorithme de Complétion SKB.* Technical Report, LRI, Orsay, France, 1988. Submitted.

[HR87] J. Hsiang and M. Rusinowitch. On word problem in equational theories. In *Proceedings of 14th International Colloquium on Automata, Languages and Programming*, Springer-Verlag, Karlsruhe (West Germany), 1987.

[Hue80] G. Huet. *A complete proof of correctness of the Knuth-Bendix completion algorithm.* Technical Report 25, INRIA, August 1980.

[Hue81] G. Huet. A complete proof of correctness of the Knuth and Bendix completion algorithm. *Journal of Computer Systems and Sciences*, 23:11–21, 1981.

[KB70] D. Knuth and P. Bendix. *Simple Word Problems in Universal Algebra*, pages 263–297. Pergamon Press, 1970.

[Kir84] H. Kirchner. A general inductive completion algorithm and application to abstract data types. In R. Shostak, editor, *Proceedings 7th international Conference on Automated Deduction*, pages 282–302, Springer-Verlag, Napa Valley (California, USA), 1984.

[KS83] K. Kapur and G. Sivakumar. Experiments with an architecture of RRL, a rewrite rule laboratory. In *Proc. of an NSF Workshop on the Rewrite Rule Laboratory*, pages 33–56, 1983.

[Les83] P. Lescanne. Computer Experiments with the REVE Term Rewriting Systems Generator. In *Proceedings, 10th ACM Symposium on Principles of Programming Languages*, ACM, 1983.

[Ore83] F. Orejas. Good food considered helpful. *Bulletin of EATCS*, 20:14–22, June 1983.

A Introduction to completion procedures

Let us take a simple example namely the type *Lists* where the constructors are $[\,], [\_], a, b$ and @ and satisfy the relations

$$[\,]@x \rightarrow x$$
$$x@[\,] \rightarrow x$$
$$(x@y)@z \rightarrow x@(y@z)$$

and a function *flatten* is given by:

$$flatten([\,]) \rightarrow [\,]$$
$$flatten(a) \rightarrow a$$
$$flatten(b) \rightarrow b$$
$$flatten(a@x) \rightarrow a@flatten(x)$$
$$flatten(b@x) \rightarrow b@flatten(x)$$
$$flatten([x]@y) \rightarrow flatten(x)@flatten(y)$$

The term $flatten([x]@[\,])$ can be rewritten into $flatten([x])$ by the second rule and into $flatten(x)$ by three rewrites, namely to $flatten(x)@flatten([\,])$ by the last rule, to $flatten(x)@[\,]$ by the fourth rule and to $flatten(x)$ by the second rule. $flatten([x]@[\,])$ is called a *superposition* and $\langle flatten(x)@flatten([\,]), flatten([x])\rangle$ a *critical pair*. If both parts of the critical pair rewrite to the same terms, the critical pair is said *convergent*, otherwise it is said *divergent*.

$$\langle (x_1@(x_2@y))@z, (x_1@x_2)@(y@z)\rangle$$

is a convergent critical pair and

$$\langle flatten(x)@flatten([\,]), flatten([x])\rangle$$

is a divergent critical pair. A completion procedure is a way to generate a rewrite system without such divergent critical pairs with the same proving power. It is based on inference

rules like the following ones where one works on a data structure with two sets, namely E that contains the identities and R that contains the rules or oriented identities.

$$\begin{array}{ll}
\textit{Delete:} & E \cup \{s = s\}; R \vdash E; R \\
\textit{Compose:} & E; R \cup \{s \to t\} \vdash E; R \cup \{s \to u\} \text{if } t \to_R u \\
\textit{Simplify:} & E \cup \{s = t\}; R \vdash E \cup \{s = u\}; R \text{ if } t \to_R u \\
\textit{Orient:} & E \cup \{s = t\}; R \vdash E; R \cup \{s \to t\} \text{ if } s > t \\
\textit{Collapse:} & E; R \cup \{s \to t\} \vdash E \cup \{u = t\}; R \text{ if } s \to_R u \text{ by a rule} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad l \to r \in R \text{ with } s \rhd l \\
\textit{Deduce:} & \qquad E; R \vdash E \cup \{s = t\}; R \text{ if } s \leftarrow_R u \to_R t \text{ for some } u
\end{array}$$

Delete removes trivial identities from E. *Compose* reduces the right-hand side of a rule if it can be rewritten by a rule in R. *Simplify* simplifies an identity. *Orient* transforms an identity into a rule provided the left-hand side is greater than the right-hand side for a given ordering. *Collapse* transforms an identity into a rule when the left-hand side is rewritten. *Deduce* creates new identities from superpositions.

The inference rule are used as long as they apply and the procedure can stop because E is empty and no rule applies or can stop with failure when no identity can be oriented or can run forever. It is *complete* if given an identity $a = b$ to be proved there exists a step i such that the R_i-normal form of a is equal to the R_i-normal form of b, where R_i is the value of R at i^{th} step. Under some assumptions of fairness not given here the procedure is complete.

B Original description of the Knuth-Bendix procedure

The next paragraph is a strict quotation of the Knuth-Bendix paper [KB70]. I found intersting to give the actual description of the algorithm we work on for now close to two decades. The corollary which is mentioned describe the concept of critical pair and (6.1) shows the stability of the congruence generated by a set of identities after adjunction of an equational consequence.

> The following procedure may now be used to attempt to complete a given set of reductions.
>
> Apply the tests of the corollary to Theorem 5, for all λ_1, λ_2 and μ. If in every case $\sigma_0' = \sigma_0''$, R is complete and the procedure terminates. If some choice of λ_1, λ_2, μ leads to $\sigma_0' \neq \sigma_0''$, then either $\sigma_0' > \sigma_0''$, $\sigma_0'' > \sigma_0'$ or $\sigma_0' \sharp \sigma_0''$. In the latter case, the process terminates unsuccessfully, having derived an equivalent $\sigma_0' \equiv \sigma_0''(R)$ for which no reduction [...] can be used. In the former cases, we add a new reduction (σ_0', σ_0'') or (σ_0'', σ_0'), respectively, to R and begin the procedure again.
>
> Whenever a new reduction (λ', ρ') is added to R, the entire new set R is checked to make sure it contains only irreducible words. This means, for each reduction (λ, ρ) in R we find irreducible words λ_0 and ρ_0 such that $\lambda \xrightarrow{*} \lambda_0$ and $\rho \xrightarrow{*} \rho_0$, with respect to $R - \{(\lambda, \rho)\}$. Here it is possible that $\lambda_0 = \rho_0$ in which case by (6.1) we may remove (λ, ρ) from R. Otherwise we might have $\lambda_0 > \rho_0$ or $\rho_0 > \lambda_0$, and (λ, ρ) may be replaced by (λ_0, ρ_0) or (ρ_0, λ_0) respectively [...]. We might also find that $\lambda_0 \sharp \rho_0$, in which case the procedure terminates unsuccessfully as above.

C Some basic notions of *CAML*

CAML is a polymorphic functional language of the *ML* family. Its basic constructions used here are the following.

let introduces an identifiers and its definition by a subexpression that will replace each occurrence of the identifier in the body that follows and which is introduced by **in**.

where is similar to **let**, but is placed after the body.

match *pattern* **with** identifies a structure that will be checked for a use as a rewrite system in the part that follows the *with*. Each rule is introduced by a pattern and the corresponding computation follows the sign $->$. The rules are separated by signs | and are evaluated with a priority according to their position. In a pattern, the sign _ means any value. For instance, (_ :: _) matches any non empty list and [_] matches any list with one element. The empty list is [].

failwith signals an exception to the a caller, such an exception is caught by a ?.

A full description appears in [FOR87b,FOR87a].