Encapsulation for Practical Simplification Procedures*

Olga Shumsky Matlin and William McCune

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

ACL2 was used to prove properties of two simplification procedures. The procedures differ in complexity but solve the same programming problem that arises in the context of a resolution/paramodulation theorem proving system. Term rewriting is at the core of the two procedures, but details of the rewriting procedure itself are irrelevant. The ACL2 *encapsulate* construct was used to assert the existence of the rewriting function and to state some of its properties. Termination, irreducibility, and soundness properties were established for each procedure. The availability of the encapsulation mechanism in ACL2 is considered essential to rapid and efficient verification of this kind of algorithm.

1 Introduction and Problem Description

We examine simplification procedures that arise in resolution, paramodulation, and rewriting systems. We have a programming problem, and at an abstract level we have a straightforward procedure to solve it. However, our theorem provers (e.g., Otter [3]) are written in C, with lots of hacks and optimizations that impose constraints that do not fit with our abstract solution. We have devised a two-stage procedure intended to have properties similar to those of the straightforward procedure. The two-stage procedure obeys the constraints, but its correctness is not obvious, so we have called on ACL2 [2] for assistance.

The following simplification problem is faced by many resolution/paramodulation style theorem-proving programs. Suppose we have a set S of clauses with the irreducibility property that no clause in S simplifies any other clause in S. We wish to add a new set I of clauses to S and have the resulting set be equivalent to $S \cup I$ and also satisfy the irreducibility property. The problem is interesting because, in addition to members of S simplifying members of I, members of I can also simplify members of S, and those simplified members can simplify other members of S, and so on. Consider the following procedure, which we call *direct incorporation*.

```
Q = I;
While (Q) do
    C = dequeue(Q);
    C = simplify(C, S);
    if (C != TRUE)
        for each D in S simplifiable by C
            move D from S to Q;
        append C to S;
```

In the terminology of our theorem prover Otter, the statement "C = simplify(C, S)" corresponds to both forward rewriting and forward subsumption, and the loop "for each D ..." corresponds to back subsumption and back rewriting. The list I represents a set of clauses derived by some inference rule.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

The direct incorporation procedure does not suit our purposes, however. The set *I* can be too large to generate in full before incorporating it into *S*. Members of *I* will typically simplify many other members of *I*, so we wish to incorporate *I* into *S* as *I* is being generated. Furthermore, the set *I* is generated by making inferences from members of *S*, and our algorithms and data structures do not allow us to remove clauses from *S* while it is being used to make inferences.

Therefore, we use a two-stage procedure, which we call *limbo incorporation*. The first stage simplifies members of *I* and, if they are not simplified to *TRUE*, puts them into a queue *L* (called the *limbo list*). The set *S* is not modified by the first stage. The second stage processes *L* until it is empty. For each member *B* of *L*, all clauses in *S* that can be simplified by *B* are removed from S, simplified by $S \cup L$, then appended to *L*. The second stage is similar to the direct incorporation procedure except that in the second stage, members of the queue being processed have already been simplified with respect to *S*. In Otter terminology, the first stage does forward simplification, and the second stage does back simplification.

The direct incorporation procedure and the limbo incorporation procedure do not necessarily produce the same results because the simplification operations can happen in different orders and the simplifiers we use do not necessarily produce unique canonical forms.

Our goals are to show, for each incorporation procedure, that (1) it terminates, (2) it produces a set in which no member can simplify any other member, and (3) the final set *S* is equivalent to the conjunction of *I* and the initial set *S*.

2 ACL2 Solution

The reasoning we need to do is primarily about the order in which simplification operations occur and the sets of simplifiers that are applied. The details of the basic simplification procedure and of the evaluation procedure for proving equivalence properties are irrelevant. Therefore we have used an ACL2 encapsulation mechanism to assert the existence and relevant properties of the simplification and evaluation functions.

An alternative to using the encapsulation mechanism is to fully define the simplification and evaluation functions and then prove the required properties based on these formalizations. Term rewriting, which is at the core of the simplification procedure, is not a simple algorithm [1], however, and considerable effort would have been required to establish its termination and necessary properties. Formalizing an evaluation function would have necessitated formalization of first-order logic in ACL2, as was done in the IVY [4] project. Our experiences in that project highlighted the difficulties in implementing a general first-order evaluation function in ACL2 and reasoning about it. Had we taken this route here, the majority of effort would have been spent on these underlying concepts, precluding us from examining the procedures of interest quickly and efficiently. For these reasons, we believe that the encapsulation mechanism was invaluable in our current work.

2.1 Constrained Functions and Their Properties

We constrain four functions using the *encapsulate* construct. The function *simplify (x y)* is for simplification of an element *x* by a set *y*. The function *true-symbolp (x)* is a recognizer for the true symbol (for example, *T* or *'true* or 1) in a particular logic. The function *ceval (x i)* is for evaluation of a clause *x* in interpretation *i*. The function *scount (x)* is for computing the size of the argument. Witnesses for the four functions are straightforward. A witness for *simplify (x y)* returns *x*. Witnesses for *true-symbolp (x)* and *ceval (x i)* always return *t*. *Acl2-count (x)* serves as a witness for *scount (x)*.

Given the witnesses, the following constraints for these four functions are stated and proved. Constraints fall into three categories depending on which of the three main goals — termination, irredicibility, and logical equivalence — they enable us to establish. To ensure termination of simplification procedures, in practice we typically use the lexicographic path ordering or the recursive path ordering [1]. Simplification with these orderings can increase the number of symbols, so *acl2-count* does not produce an accurate termination function. Instead, the constrained function *scount* is used to determine the size of a clause. The main property of the function is that it returns a natural number.

```
(defthm scount-natural
  (and (integerp (scount x))
       (<= 0 (scount x))))
```

Termination proofs depend on the constraint that for formulas that are indeed changed by simplification, the result of the simplification is somehow smaller than the original expression.

```
(defthm scount-simplify
  (or (equal (simplify x y) x)
      (< (scount (simplify x y))
         (scount x))))
```

Proof of the irreducibility property depends on the following properties of the basic simplification procedure. An idempotence property states that once a formula is simplified by a set, attempting to simplify the result again by the same set will have no effect. Another property requires that if a set simplifies a formula, then a superset of that set does so as well. A third property states that two sets that do not simplify a formula individually do not do so when considered collectively.

```
(defthm simplify-idempotent
  (equal (simplify (simplify x y) y)
         (simplify x y)))

(defthm simplify-subset
  (implies (and (not (equal (simplify a x) a))
                (subsetp-equal x y))
           (not (equal (simplify a y) a))))

(defthm simplify-append
  (implies (and (equal (simplify a x) a)
                (equal (simplify a y) a))
           (equal (simplify a (append x y)) a)))
```

We formalized the notion of rewritability to improve the readability of the formalizations of both the direct and limbo incorporation procedures and to ease management of proofs. If a set simplifies an element, we say that the element is rewritable by the set. The new function *rewritable* is defined outside the encapsulation. Once the termination and irreducibility constraints are restated in terms of *rewritable*, the function is disabled.

```
(defun rewritable (x y)
  (not (equal (simplify x y) x)))
```

Finally, the proofs of the logical equivalence property of our incorporation procedures depend on the following properties of the constrained evaluation function and its relationship with *simplify* and *true-symbolp*. The evaluation function is Boolean, and the true symbol of the logic is evaluated to true. We define a function to evaluate a set of elements as a conjunction. The main soundness property of constrained simplification states that if the conjunction of simplifiers is true, the evaluations of the original and simplified expressions are equal.

```
(defthm ceval-boolean
  (or (equal (ceval x i) t) (equal (ceval x i) nil)))

(defthm true-symbolp-ceval
  (implies (true-symbolp x) (ceval x i)))

(defun ceval-list (x i)
  (if (endp x)
      t
    (and (ceval (car x) i) (ceval-list (cdr x) i))))

(defthm simplify-sound
  (implies (ceval-list y i)
           (equal (ceval (simplify x y) i) (ceval x i))))
```

2.2 Formalization and Termination of Incorporation Procedures

Three supporting functions are used to formalize the direct and limbo incorporation procedures. Rather than present the ACL2 implementation of the functions, we simply describe them. The function *extract-rewritables (x s)* computes a subset of elements of *S* that are rewritable by *X*. The function *extract-n-simplify-rewritables (x s)* produces a set of elements of *S* that are rewritable by *X* and have been simplified by it. The function *remove-rewritables (x s)* produces the set of elements of *S* that are not rewritable by *X*. The direct incorporation procedure is formalized by using the last two functions as follows.

```
(defun direct-incorporation (q s)
  (cond ((or (not (true-listp q)) (not (true-listp s))) 'INPUT-ERROR)
        ((endp q) s)
        ((true-symbolp (simplify (car q) s)) (direct-incorporation (cdr q) s))
        (t (direct-incorporation
            (append (cdr q)
                    (extract-n-simplify-rewritables (simplify (car q) s) s))
            (cons (simplify (car q) s)
                  (remove-rewritables (simplify (car q) s) s))))))
```

The limbo incorporation procedure relies on computation of the initial limbo list and subsequent integration of the list into the original database. As stated above, the second step of the incorporation procedure may place new elements on the limbo list. Before any element is added to the limbo list, however, it is simplified as much as possible by the members of the original database and the elements already on the limbo list. We note, therefore, that in the recursive call of the function *preprocess-list*, in addition to the the members of original database and limbo list, the set of simplifiers includes elements processed by the function in the previous calls.

```
(defun preprocess (x s l)
  (if (true-symbolp (simplify x (append s l)))
      l
    (append l (list (simplify x (append s l))))))

(defun initial-limbo (q s l)
  (if (endp q)
      l
    (initial-limbo (cdr q) s (preprocess (car q) s l))))

(defun preprocess-list (d s r)
  (if (endp d)
      r
      (preprocess-list (cdr d) s (preprocess (car d)
                                             (append s (cdr d))
                                             r))))

(defun process-limbo (l s)
  (cond ((or (not (true-listp l)) (not (true-listp s))) 'INPUT-ERROR)
        ((endp l) s)
        (t (process-limbo (append (cdr l)
                                  (preprocess-list
                                   (extract-rewritables (car l) s)
                                   (append (remove-rewritables (car l) s) l)
                                  nil))
                          (cons (car l)
                                (remove-rewritables (car l) s))))))

(defun limbo-incorporation (q s)
  (process-limbo (initial-limbo q s nil) s))
```

Termination proofs for the functions *direct-incorporation* and *process-limbo* rely on the simplification properties stated in the encapsulation. The proofs are not entirely trivial; in order to achieve them, the conjectures must be split into two cases: a case when the set of elements produced by the *extract* functions is empty, and a case when it is not. We define an additional counting function *lcount* whose behavior on lists is similar to that of *acl2-count*, except that the size of list elements is computed by using the constrained function *scount*.

```
(defun lcount (x)
  (if (endp x)
      0
     (+ 1 (scount (car x)) (lcount (cdr x)))))
```

The measure function, based on *lcount*, is

```
(cons (+ 1 (lcount q) (lcount s))
      (+ 1 (lcount q))).
```

We note that the formalization on the direct incorporation procedure is slightly different from the algorithm presented in Section 1. In the algorithm elements D that are rewritable by C are moved from the set S onto Q. In the formalization, these elements are simplified by C before being placed onto Q. This extra simplification step allows us to show that the direct incorporation algorithm terminates. Yet this addition to the original algorithm does not affect the main correctness properties of the procedure.

2.3 Irreducibility Property

We formulate the irreducibility property as follows. We first define a function *mutually-irreducible-el-list (x s)* that checks that the element *X* neither rewrites nor is rewritable by anything in *S*. The main irreducibility check function relies on the element wise irreducibility check.

```
(defun mutually-irreducible-el-list (x s)
  (cond ((endp s) t)
        ((or (rewritable x (list (car s)))
             (rewritable (car s) (list x))) nil)
        (t (mutually-irreducible-el-list x (cdr s)))))

(defun irreducible-list (s)
  (cond ((endp s) t)
        ((mutually-irreducible-el-list (car s) (cdr s))
         (irreducible-list (cdr s)))
        (t nil)))
```

We accomplished the second of the stated goals by proving that if the original database of clauses is irreducible, both incorporation procedures produce sets with that property.

```
(defthm direct-incorporation-is-irreducible
  (implies (irreducible-list s)
           (irreducible-list (direct-incorporation q s))))

(defthm limbo-incorporation-is-irreducible
  (implies (irreducible-list s)
           (irreducible-list (limbo-incorporation q s))))
```

2.4 Soundness

Soundness proofs rely on the properties of ceval given in the *encapsulate* construct and were relatively easy to establish. We showed that both incorporation procedures produce a conjunction of clauses whose evaluation is equivalent to the evaluation of the conjunctions of clauses in the two input sets.

```
(defthm direct-incorporation-is-sound
  (implies (and (true-listp q)
                (true-listp s))
           (equal (ceval-list (direct-incorporation q s) i)
                  (and (ceval-list q i) (ceval-list s i)))))

(defthm limbo-incorporation-is-sound
  (implies (true-listp s)
           (equal (ceval-list (limbo-incorporation q s) i)
                  (and (ceval-list q i) (ceval-list s i)))))
```

3 Related Work and Conclusions

Our earlier project IVY [4] dealt with checking the proofs produced by Otter. The checker code was written in ACL2 and proved sound. Although both efforts concern the same software, the errors they help eliminate do not overlap. IVY was designed to catch errors in Otter-produced proofs. This work focuses on irreducibility and termination, and errors in the simplification procedure described here would likely not lead to soundness problems in the resulting proofs, but would prevent Otter from finding some or all proofs for a particular problem.

Also related is the large and ongoing ACL2 effort on abstract reduction systems and term rewriting in [5]. The effort concentrates on formalizing basic reduction and rewriting procedures in ACL2 and establishing their properties. The work includes formalization of first-order logic and reasoning about termination of rewriting. Both are aspects that our effort takes for granted to concentrate on a practical application that relies on a rewriting procedure.

The Otter code is based on an algorithm similar to limbo incorporation. Correctness of this algorithm is therefore important to us but is not obvious because of the complexity of the algorithm. While the algorithm depends on term rewriting and clause subsumption procedures, we were able, thanks to encapsulation mechanism in ACL2, to concentrate on only a few relevant properties of these basic procedures and to devote all effort to understanding and verifying the limbo incorporation, the actual procedure of interest.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, United Kingdom, 1998.

[2] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[3] W. McCune. Otter 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994. See also URL http://www.mcs.anl.gov/AR/otter/.

[4] W. McCune and O. Shumsky. IVY: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 16. Kluwer Academic, 2000.

[5] J. L. Ruiz Reina, J. A. Alonso, M. J. Hidalgo, and F. J. Martín. Formal proofs about rewriting using ACL2. *Annals of Mathematics and Artificial Intelligence*, 36(3):239–262, 2002.