

1 Announcements

1. Exam on Thursday will cover everything up until the end of the day today.
2. The last homework is up today. It will required that develop a video game, lights out. There is plenty of extra credit.
3. The plan for the rest of the semester follows. On Wed we will have a lecture on graphics in ACL2s. Useful for you last homework, so read about it and come prepared with questions. Thursday is an exam. Next week we meet on Monday and Wednesday and we will cover set theory and logic.
4. The final for section 1 (10:30-11:35) is on April 18 at 1PM. That is the Friday rifht after classes end; that's two days after the last class. The final for section 2 is on April 23 at 1PM. That is one week after classes end, on a Wednesday. See Web page for details.

2 Review

Last time we discussed in more depth rewriting. The summary is that theorems give rise to rewrite rules and these rules have a lhs (left-hand side) and a rhs (a right-hand side). So, it is important when we state theorems that we take this into account and we *orient* the rules so that lhs is more complex than the rhs. Every time you state a theorem you *have to* think about the rewrite rule you get. At a higher level, you should aim to use rewrite rules to drive expressions into a “canonical” form.

We also got experience in determining what to do when proof attempts fail. The rough idea is to look at checkpoints and to use our model of how ACL2 works to come up with a plan that will allow ACL2 to make progress. We also saw that one of the issues is that you can get into infinite loops.

Finally, we looked at the `defunt` macro, which is on the Web and saw that you should be using that to define your functions, since it allows you to specify the input and output types of your functions, which in turn leads to ACL2 proving theorems.

3 Using ACL2

Today, we are going to look at three other aspects of ACL2: propositional reasoning, congruence closure, and equivalences.

3.1 Simplification

Rewriting is part of simplification, the first proof technique. More than 90% of the interesting technology in ACL2 is in the simplifier. Let's look at another aspect of simplification, decision procedures. Here we will look at two decision procedures.

What are decision procedures? Why are they desirable?

The two decision procedures we will look at are propositional calculus and congruence closure, since I think they will help us be effective users of ACL2s.

3.1.1 Propositional Calculus

The basic propositional calculus decision procedure is based on the normalization of `if` expressions:

1. propositional connectives are expanded in terms of `if`;
2. the `if` terms are distributed, so $(f \text{ (if } a \text{ } b \text{ } c))$ becomes $(\text{if } a \text{ (} f \text{ } b \text{ (} f \text{ } c))$ and $(\text{if (if } a \text{ } b \text{ } c) \text{ } x \text{ } y)$ becomes $(\text{if } a \text{ (if } b \text{ } x \text{ } y) \text{ (if } c \text{ } x \text{ } y))$; and
3. the resulting tree is explored to determine whether every reachable tip is `non-nil`.

```
(thm (iff (implies a b) (or (not a) b)))
```

```
(thm (iff (and a b) (or (not a) (not b))))
```

```
(thm (iff (and a b) (iff a (iff b (or a b)))))
```

3.1.2 Congruence Closure

The congruence closure procedure uses the context to compute equivalence classes, chooses a canonical representative of every equivalence class, and substitutes that representative for all members of the class into all function applications allowing it. For example, if $(= a b)$ and $(= b c)$ are known from the present context, then $(= a c)$ is added to the context; and moreover, if a is the canonical representative of its class, then occurrences of b and c are replaced by a . This is done iteratively.

This also works for equivalence relations beyond equality.

For example, consider the following.

We start by defining two functions, `f` and `g`, that take one argument and return one arguments, but about which we know nothing else.

```
(defstub f (*) => *)
```

```
(defstub g (*) => *)
```

Consider the following query. Is it true? Since this is the first time we have seen non-primitive functions for which we do not have a definition, what do we mean when we ask whether a query such as the one below is true? For example, what would a counter-example look like? A counterexample would provide us with a value for x , as expected, but also values for the functions f and g . So, the queries are true if they are true for every possible pair of functions, f and g .

```
(thm (implies (and (= (f (f x)) x)
                    (= (f (f (f x))) x))
            (= (g (f x))
              (g x))))
```

```
(thm (implies (and (= (f (f x)) x)
                    (= (f (f (f x))) x))
            (= (f (g x))
              (g x))))
```

```
(thm (implies (and (= b c)
                    (= (g (f c)) a)
                    (not (= a (g b))))
            (not (= (f b) c))))
```

Which of the above are true? Use ACL2. If they aren't true, come up with a counterexample.

4 Congruence-Based Reasoning

This section is mostly taken directly from our class textbook.

The most frequently used rule of inference in most proofs is the familiar idea of substitution of equals for equals. ACL2 supports a general form of substitution of equals for equals, based on the ideas of user-defined equivalence relations and congruence rules. The importance of user-defined equivalence and congruence rules is difficult to appreciate at first. They inherit their importance, in part, from the fact that ACL2 does not provide abstract data types. New kinds of objects must be represented in terms of existing ACL2 primitives, *e.g.*, sets are represented as lists. The operations on these objects are defined as functions on the existing data types, *e.g.*, the set operations are functions on lists that ignore duplication and order. Because such representations are often not unique, ACL2's equality predicate, `equal`, is too strong: it distinguishes objects that all the operations of the new type treat equivalently. Thus, one must define an equivalence relation on the new type and prove that the new operations respect this notion of equivalence. Once that is done, ACL2 can use that equivalence to substitute into nests of the new operations. Use of equivalence relations and congruence rules is fundamental to the simplifier as well as other proof techniques in the waterfall.

The need for equivalence relations is easily seen by considering the representation of finite sets as linear lists. Thus, we might think of (1 2 3) and (3 1 2 1) as equivalent representations of the set {1, 2, 3}.

Suppose we define (`un a b`) to return (a representation of the) union of (the sets represented by) `a` and `b`. Thus, (`un '(1 2) '(3 4)`) might be (1 2 3 4). Is `un` commutative? Depending on how we actually define it, it may not be commutative. For example, (`un '(1 2) '(3 4)`) might be (1 2 3 4) but (`un '(3 4) '(1 2)`) might be (3 4 1 2). These two objects are not `equal`. But we could define a relation, (`set-equal a b`), that returns `t` or `nil` according to whether the set represented by `a` is the same as the set represented by `b`. Then we would have the theorem

```
(set-equal (un b a) (un a b)).
```

Suppose we use `mem` to test for “set membership,” *i.e.*, (`mem e x`) is `t` or `nil` according to whether `e` is an element of `x`. Consider proving

```
(implies (mem e (un a b))
         (mem e (un b a))).
```

The “natural” proof is “use the commutativity of `un` to replace (`un b a`) by (`un a b`).” If asked to justify such a move, we might say “substitution of equals for equals.” But the commutativity result we have for `un` is not an equality! What allows us to use it to replace (`un b a`) by (`un a b`) in our conjecture?

One might respond by observing that `set-equal` is an equivalence relation: it is symmetric, reflexive, and transitive. While true, that is not enough. For example, we cannot use commutativity to replace (`un b a`) by (`un a b`) in (`equal (car (un a b)) (car (un b a))`) or else we could prove a non-theorem. It is important that `mem` respects `set-equal` in the sense that (`mem e x`) returns the same result as (`mem e y`) whenever `x` is `set-equal` to `y`. This is a *congruence rule* and might be phrased as

```
(implies (set-equal x y)
         (equal (mem e x) (mem e y))).
```

This congruence rule allows us to substitute `set-equals` for `set-equals` in the second argument of a `mem` expression, without changing the value of the `mem` expression. Given the congruence rule, we can use the commutativity rule—as stated in terms of `set-equal`—just as though it were an equality. That is, we can use it as a rewrite rule.

There is one final twist to discuss. The twist has to do with the fact that the congruence rule uses two different senses of equality. In the rule above we see both `set-equal` and `equal`. In general, we might see any two equivalence relations here. For example, in Lisp the membership “predicate” is named `member`. But instead of returning `t` to indicate success, (`member e x`) returns the first tail of `x` that starts with `e`. This way `member` can be used to determine both whether and where `e` occurs in `x`. For this sense of membership, the congruence rule above does not hold. Let `x` be '(1 2) and `y` be '(2 1). Let `e` be 1. Then (`set-equal x y`) is true. But (`member e x`) returns (1 2) while (`member e y`) returns (1). These two objects are not `equal`. But they are

“propositionally equal” in the sense that they are `nil` or `non-nil` in unison. So we have another congruence rule.

```
(implies (set-equal x y)
         (iff (member e x) (member e y)))
```

This congruence rule allows the substitution of `set-equals` for `set- equals`, in the second argument of `member` expressions, while preserving `iff`. Put another way, if a `member` expression occurs in a position in which only its propositional value is important, then its second argument occurs in a position in which only its set value is important.

At the top of a conjecture, only the propositional value is important. As we explore the subterms occurring in the conjecture, we can use congruence rules to keep us apprised of which equivalence relations we must preserve to maintain top-level propositional equivalence.

A binary relation is known to be an equivalence relation if it has been proved Boolean, symmetric, reflexive, and transitive and those facts have been marked as an `equivalence` rule. To prove and store the necessary rules, use the `defequiv` command.

We say an occurrence of *lhs* in a formula is *equiv-hittable* if congruence rules are available to establish that the occurrence can be replaced by any equivalent (modulo *equiv*) term without changing the propositional value of the formula.

Congruence rules are derived from theorems of the form

```
(implies (equiv1 x y)
         (equiv2 (f ... x ...)
                  (f ... y ...)))
```

where *equiv₁* and *equiv₂* are known equivalence relations. To prove a congruence rule, use `defcong` and see also `congruence`. Such a congruence rule informs ACL2 that if *x* and *y* are equivalent (modulo *equiv₁*) then the result of replacing one by the other in the indicated argument position of *f* produces an equivalent term (modulo *equiv₂*). We sometimes characterize such a congruence by saying that it allows *equiv₁* substitution (in the indicated argument position) into *f* while *preserving equiv₂*. One can thus justify a deep substitution by chaining together congruence rules, starting from a congruence rule that preserves `iff` (propositional equivalence). ACL2 can do such chaining, provided appropriate congruence rules are available for every relevant argument position of every relevant function symbol.

Generally speaking when you represent a new type of object (*e.g.*, sets as lists) you might consider introducing a corresponding equivalence relation. Then when you define the elementary operations on the “new” objects, *e.g.*, `mem` and `un`, you should consider proving the appropriate congruence rules. Here are the rules for `mem` and `un`.

```
(implies (set-equal x y)
         (iff (mem e x)
              (mem e y)))
```

```

(implies (set-equal x y)
         (set-equal (un x a)
                    (un y a)))
(implies (set-equal x y)
         (set-equal (un a x)
                    (un a y)))

```

The first says that propositional equivalence is preserved when set equivalence is preserved in the second argument of `mem`. The second says that set equivalence is preserved when set equivalence is preserved in the first argument of `un`. The third says that set equivalence is preserved when set equivalence is preserved in the second argument of `un`. These three rules may be conveniently expressed as shown below. The names of the variables are unimportant.

```

(defcong set-equal iff (mem e x) 2)
(defcong set-equal set-equal (un x a) 1)
(defcong set-equal set-equal (un a x) 2)

```

`Defcong` is defined as a macro that expands into a `defthm` form. This is a common use of macros. See `defcong`.

Now suppose that the rewriter encounters the term

```

(mem  $\alpha$ 
     (un (un  $\beta$   $\gamma$ )  $\delta$ ))

```

while it is trying to preserve propositional equivalence. The three congruence rules tell the rewriter that it can replace β , γ , and δ (as well as the `un`-terms containing them) by `set-equal` terms. You should think of congruence rules as providing a road-map with which ACL2 can figure out the equivalence relations to preserve while rewriting given subterm occurrences.

Why bother? The knowledge that it is sufficient to preserve `set-equal` while rewriting, say, β , is only important if you have also proved rules that allow the rewriter to replace one term by a `set-equal` term. Here are some such rules.

```

(set-equal (un b a) (un a b))
(set-equal (un (un a b) c) (un a (un b c)))
(set-equal (un b nil) b)

```

These are just rewrite rules; they direct the system to replace the left-hand side by the right-hand side in `set-equal`-hittable contexts.

Congruence rules just permit these rules to be used.