Peter Dillinger

Theorems and Theories

Is the following a theorem?

```
        (implies (> x 20000)
                 (too-big x))
```

Well, if we have not defined the function `too-big`, then it certainly is not a theorem and ACL2 won't even attempt to prove the proposition.

If we make this definition,

```
        (defun too-big (x)
          (> x 1000))
```

then the theorem is clearly true. ACL2 proves it:

```
        (thm (implies (> x 20000)
                      (too-big x)))
```

But if we define `too-big` differently,

```
        (defun too-big (x)
          (> x 1000000))
```

it is not a theorem. Here's a counterexample:

```
        (check=
         (let ((x 30000))
           (implies (> x 20000)
                    (too-big x)))
         nil)
```

The point here is that whether a proposition is a theorem or not depend on the current *theory*, which depends on all the function definitions.

The Definitional Principle

Let's look at the requirements for function definitions in ACL2 and what they mean in the ACL2 logic. This is given by the **definitional principle**:

A function definition, (defun *f* (v_1 v_2 ... v_n) *body*) is **admissible** if

- f is a new function symbol, meaning it has not yet been defined
- v_1 v_2 ... v_n are distinct variable symbols
- *body* is a valid ACL2 expression, possibly calling f recursively, and no free variables other than v_1 v_2 ... v_n
- ACL2 is able to prove the function terminates on all inputs (see comments below)

If admissible, the definition is accepted, meaning it can be used and executed in code and adds a new axiom to the current logical theory:

- `(equal (`f v_1 v_2 … v_n`)` *body*`)`

We will talk about axioms and theories soon, but let us talk about termination again. Recall examples from last lecture in which functions that did not terminate caused trouble for the logic, because they did not corresponded to total, mathematical functions. Consequently, ACL2 only accepts function definitions that are proven to terminate. The textbook on ACL2 covers in depth how the user can prove functions terminating in ACL2, but we will not cover that in this class. We will depend on a system referred to as “CCG termination analysis” for proving termination automatically. It is built into the ACL2s “Intermediate” session mode, among others.

If the CCG termination analysis succeeds, it was able to prove the function terminates:

```
ACL2 >EVENT
(defun app (x y)
  (if (endp x)
    y
    (cons (car x) (app (cdr x) y))))

Attempting to prove termination using CCG analysis (please be patient)...

CCG analysis has succeeded in proving termination of APP.  Thus, we
admit this function under the principle of definition.  We observe
that the type of APP is described by the theorem
(OR (CONSP (APP X Y)) (EQUAL (APP X Y) Y)).  We used primitive type
reasoning.

Summary
Form:  ( DEFUN APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:WELL-FOUNDED-RELATION WELL-FOUNDED-L<))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other: 0.01)
```

If the CCG termination analysis fails, it could be because the function does not terminate on some inputs, but it could also be because it wasn't “smart” enough to prove termination. Here's an example on a function that does not terminate on some inputs:

```
ACL2 >EVENT
(defun fact (x)
  (if (= x 0)
    1
    (* x (fact (- x 1)))))

Attempting to prove termination using CCG analysis (please be patient)...


ACL2 Error in ( DEFUN FACT ...):  CCG analysis has FAILED to prove
termination of FACT.  Thus, we suspect (but haven't proven) that there
are some inputs for which this function does not terminate.


Our recommendations:
```

```
* Attempt to find counterexamples to termination.

* Lend some aid to the CCG analysis:
-- Attempt a simplifying redesign of this function.
-- Prove lemmas, build up some theory, and/or include books related
to functions in the proposed definition that are critical to termination.

* Provide an explicit termination argument for a measure-based proof.
See :DOC measure and :DOC defun. The default well-founded relation
is currently L<, which is lexicographic ordering on lists of natural
numbers.


Summary
Form:  ( DEFUN FACT ...)
Rules: NIL
Warnings:  None
Time:  0.08 seconds (prove: 0.08, print: 0.00, proof tree: 0.00, other: 0.00)

ACL2 Error in ( DEFUN FACT ...):  See :DOC failure.

******** FAILED ********
```

But in this class, you do not need measure-based proofs.

Theories and Axioms

The definitional principle says that a definition like

```
(defun too-big (x)
  (> x 1000))
```

adds a new axiom to the current logical theory:

```
(equal (too-big x) (> x 1000))
```

What does this mean? In fact, a logical **theory** is a set of axioms, and an **axiom** is a proposition that is *assumed* to be true. An axiom is a fundamental truth about the system we are working on. A synonym of axiom you have probably heard before is *postulate*.

What do axioms do for us? That is where a **logic** comes in, with **rules of inference**, which allow us to derive theorems from axioms and other theorems.

Now we have a picture of where proofs of theorems come from: a theorem concerns a given theory in a given logic. That theory is a set of axioms. The logic has rules of inference that allow us to generate other theorems from those axioms. (Axioms are theorems.)

When we start ACL2, it has lots of functions already defined and it correspondingly has axioms for those functions in its theory. Remember from the ACL2 Language Overview that some functions are "primitive" and some are "derived?" The axioms governing derived functions come from the definitional principle. For example, there is an axiom that says

```
(equal (true-listp x)
       (if (consp x)
         (true-listp (cdr x))
         (equal x nil)))
```

which comes from the definition of true-listp.

Primitive functions, however, are not defined in terms of other functions, so the axioms governing those were put together carefully and cleverly by the people who created ACL2. Here are some examples of those:

```
        (equal x x)
        (not (equal t nil))
        (implies (equal x nil)
                 (equal (if x y z)
                        z))
        (implies (not (equal x nil))
                 (equal (if x y z)
                        y))
```

Rules of inference

The textbook describes a small, rather cryptic set of inference rules for ACL2. There is an elegance in making a logic as simple as possible, but it can be hard to work with. I will describe a more general set of inference rules that should be easy to understand and more easily usable. Here are some:

Instantiation: If φ is a theorem, then φ with all occurences of a free variable replaced by some expression is also a theorem. We could also state this as

$$\varphi \Rightarrow \texttt{(let ((}v\ e\texttt{))}\ \varphi\texttt{)}$$

because the meaning of LET is to replace free occurrences of v with e. However, we don't want to depend on LET for this rule, so go ahead and do the replacement.

For example,

```
        (consp (cons a b))
```

is an axiom and, therefore, a theorem. We can **instantiate** `a` with any expression, including `(car x)`, and get another theorem:

```
        (consp (cons (car x) b))
```

The rule of inference tells us this must be a theorem. Furthermore, we can instantiate `b` to get yet another theorem:

```
        (consp (cons (car x) (app (cdr x) y)))
```

Propositional deduction: If

$$(\varphi_1 \land \varphi_2 \land \ldots \land \varphi_n) \rightarrow \psi$$

is a propositional tautology—a tautology according to boolean logic—and $\varphi_1, \varphi_2, \ldots, \varphi_n$ are theorems, then we can conclude ψ is a theorem.

In other words,

$$\varphi_1, \varphi_2, \ldots, \varphi_n \Rightarrow \psi \quad \text{if} \quad (\varphi_1 \land \varphi_2 \land \ldots \land \varphi_n) \rightarrow \psi \quad \text{is a tautology}$$

For example, from these ACL2 axioms,

```
        (implies (integerp x) (acl2-numberp x))
        (implies (consp x) (not (acl2-numberp x)))
```

I claim I can deduce

```
        (implies (integerp x) (not (consp x)))
```

I can get this by a propositional deduction. If you let i stand for `(integerp x)`, n for `(acl2-numberp x)` and c for `(consp x)`, the following must be a tautology for the deduction to be legal:

$$((i \rightarrow n) \wedge (c \rightarrow \neg n)) \rightarrow (i \rightarrow \neg c))$$

Well, $c \rightarrow \neg n$ is equivalent to $n \rightarrow \neg c$, and $(i \rightarrow n) \wedge (n \rightarrow \neg c)$ implies $i \rightarrow \neg c$. Thus, it is a tautology. (Check it with a truth table.) Thus, the deduction above is a legal application of propositional deduction. Thus, `(implies (integerp x) (not (consp x)))` is a theorem.

A special case of propositional deduction is **Modus ponens**. It says that if we have theorems φ and (`implies` φ ψ), then we can conclude ψ. That is because $(\varphi \wedge (\varphi \rightarrow \psi)) \rightarrow \psi$ is a tautology.

Equals for equals: If we have a theorem that e_1 is equal to e_2 and another theorem φ, then we can conclude φ with occurrences of e_1 in it optionally replaced by e_2.

For example, if we have theorems

```
        (equal (> x 1000) (too-big x))

        (implies (> x 20000)
                 (> x 1000))
```

Then we can **replace equals for equals** to conclude

```
        (implies (> x 20000)
                 (too-big x))
```

In this case, e_1 would be `(> x 1000)`, e_2 would be `(too-big x)`, and φ would be `(implies (> x 20000) (> x 1000))`.

Sample Proof

I will prove

```
        (not (consp (len x)))
```

I will assume a theorem I proved above,

```
        (implies (integerp x) (not (consp x)))
```

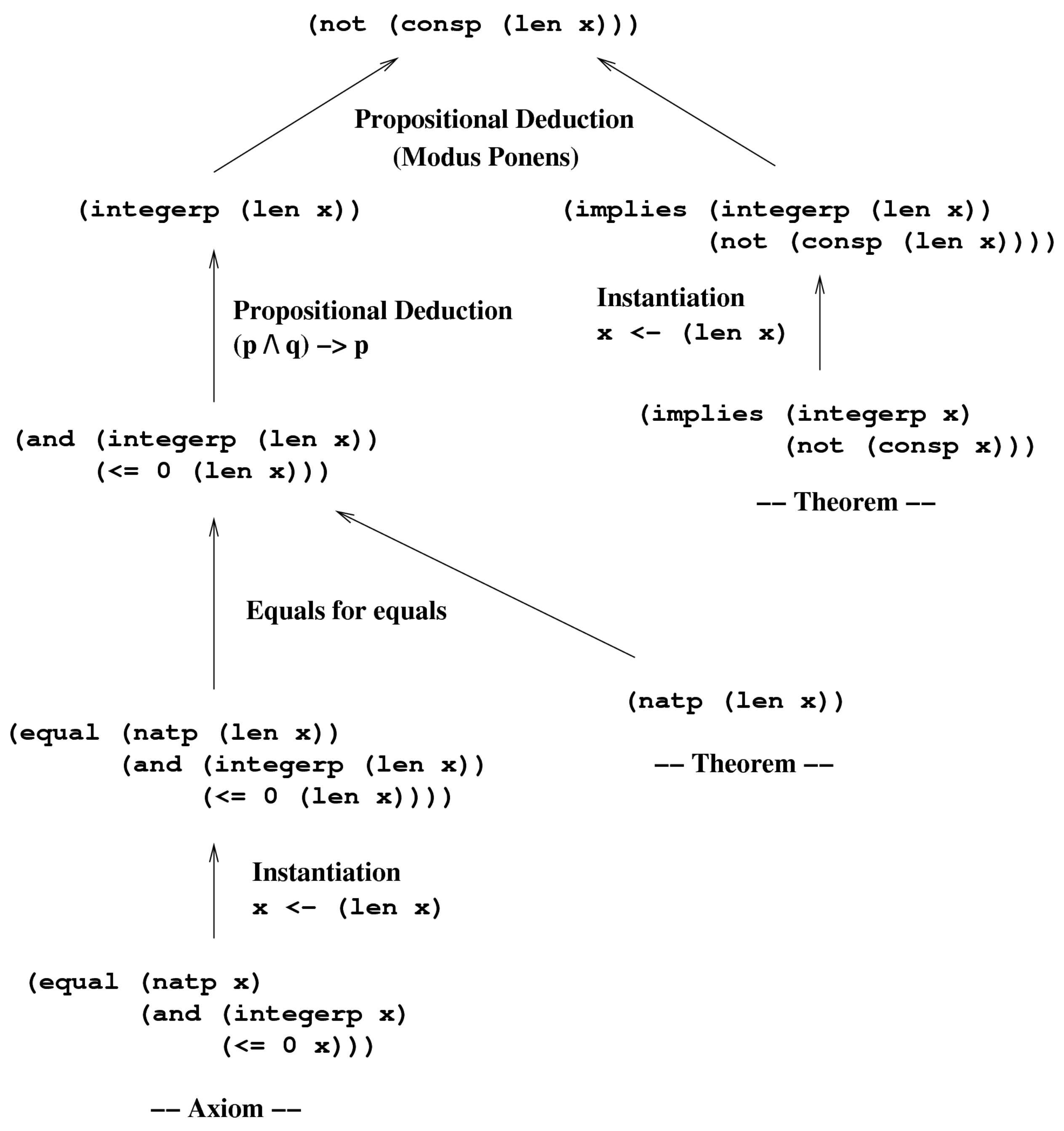
another I will not prove here,

```
        (natp (len x))
```

and the definitional axiom for NATP:

```
        (equal (natp x)
               (and (integerp x) (<= 0 x)))
```

I will, in a sense, do this proof "backwards," by starting with what we want to prove and eventually reducing to axioms or known theorems:


(not (consp (len x)))
Propositional Deduction
(Modus Ponens)
(integerp (len x))
(implies (integerp (len x)) (not (consp (len x))))
Propositional Deduction
(p Λ q) –> p
Instantiation
x <- (len x)
(and (integerp (len x)) (<= 0 (len x)))
(implies (integerp x) (not (consp x)))
–– Theorem ––
Equals for equals
(natp (len x))
–– Theorem ––
(equal (natp (len x)) (and (integerp (len x)) (<= 0 (len x))))
Instantiation
x <- (len x)
(equal (natp x) (and (integerp x) (<= 0 x)))
–– Axiom ––

More specifically, a **formal proof** is a tree of propositions whose tips ("leaves") are axioms and each other node is a consequence of its children by some rule of inference. The root node has the overall theorem that is proven.

Technically, the proof above is not a complete formal proof, because some of the tips are just known theorems rather than axioms. For it to be a complete formal proof, one would need to fill in proofs of those theorems, but it is standard to use previously proven theorems in subsequent proofs.