Using Counterexample Generation and Theory Exploration to Suggest Missing Hypotheses

Ruben Gamboa[1,2] Panagiotis Manolios[3] Eric Smith[2] Kyle Thompson[2,4]

[1]University of Wyoming

[2]Kestrel Institute

[2]Northeastern University

[4]University of California San Diego

ACL2 Workshop 2023
Austin, TX

Outline

- Context
- Introducing DrLA
- A Classic Example
- Conclusion

Goals

- Easier proof repair
  - Evolution of definitions
  - Rearranging of libraries or previous theorems
  - New versions of theorem prover

- Easier proof discovery
  - Useful libraries
  - Useful lemmas
  - Helpful hints

Goals

- Easier proof repair
  - Evolution of definitions
  - Rearranging of libraries or previous theorems
  - New versions of theorem prover

- Easier proof discovery
  - Useful libraries
  - Useful lemmas
  - Helpful hints
  - **Missing hypotheses**

PEARLS Project

- Eric will cover this better in a rump session
- Basic Idea
  - Collect many theorems from community books
  - Break them in some way (so we know the “fix”)
  - Capture the ACL2 breakpoints
  - Train an AI to match breakpoints with the “fixes”

PEARLS Project

- Eric will cover this better in a rump session
- Basic Idea
  - Collect many theorems from community books
  - Break them in some way (so we know the “fix”)
  - Capture the ACL2 breakpoints
  - Train an AI to match breakpoints with the “fixes”
- Three big issues
  - Is AI better than ad hoc suggestions?
  - Should we learn from checkpoints or counterexamples?
  - What if the “broken” theorem turns out to be false?

Outline

- Context
- Introducing DrLA
- A Classic Example
- Conclusion

DrLA: The Doctor's Logic Assistant

- Attempts to find missing hypotheses
- Only applies when theorem is false

DrLA: The Doctor’s Logic Assistant

- Attempts to find missing hypotheses
- Only applies when theorem is false

- Not based on ML, in particular not generative AI

DrLA: The Doctor's Logic Assistant

- Attempts to find missing hypotheses
- Only applies when theorem is false

- Not based on ML, in particular not generative AI

- Based on theory exploration
  1. Start with a vocabulary of function names, variables, and constants
  2. Generate a forest of expression trees
  3. Check each generated expression to see if it's useful
  4. Profit!!!

Generating Expressions

- Start with `consp`, `nil`, `cons`, `car`, `cdr`, `append`, and `equal`

Generating Expressions

- Start with `consp`, `nil`, `cons`, `car`, `cdr`, `append`, and `equal`

- Familiar theorems

```
(equal (car (cons x1 x2)) x1)
(equal (append (append x1 x2) x3) (append x1 (append x2 x3)))
```

Generating Expressions

- Start with `consp`, `nil`, `cons`, `car`, `cdr`, `append`, and `equal`

- Familiar theorems

```
(equal (car (cons x1 x2)) x1)
(equal (append (append x1 x2) x3) (append x1 (append x2 x3)))
```

- Plausible (but false) theorems

```
(equal (car (cons x1 x2)) x2)
(equal (append x1 x2) (append x2 x1))
```

Generating Expressions

- Start with `consp`, `nil`, `cons`, `car`, `cdr`, `append`, and `equal`

- Familiar theorems

```
(equal (car (cons x1 x2)) x1)
(equal (append (append x1 x2) x3) (append x1 (append x2 x3)))
```

- Plausible (but false) theorems

```
(equal (car (cons x1 x2)) x2)
(equal (append x1 x2) (append x2 x1))
```

- Utter nonsense

```
(consp (equal (car nil) (append x1 x2)))
(car (cons (cdr x1) (equal x2 nil)))
```

Testing Expressions

- Not enough to check if the suggested hypothesis can prove the theorem
- E.g., what if the suggested hypothesis is NIL?

Testing Expressions

- Not enough to check if the suggested hypothesis can prove the theorem
- E.g., what if the suggested hypothesis is NIL?

- Use counter-example generation (cgen from ACL2s)
  - produce counterexamples and witnesses to original theorem
  - test the proposed hypothesis on the counterexamples
  - ... and on the witnesses

Testing Expressions

- Not enough to check if the suggested hypothesis can prove the theorem
- E.g., what if the suggested hypothesis is NIL?

- Use counter-example generation (cgen from ACL2s)
  - produce counterexamples and witnesses to original theorem
  - test the proposed hypothesis on the counterexamples
  - ... and on the witnesses

- Original idea was that the suggested hypothesis should be
  - false for all counterexamples
  - true for all witnesses
- We found it was useful to allow the hypothesis to be false for <u>some</u> witnesses

Choosing the Language

- We need function names, variables, and constants

Choosing the Language

- We need function names, variables, and constants
- The functions come from
  - a handful of built-in functions
  - functions appearing in the theorem
  - functions appearing in the definitions of functions in the theorem
  - . . . up to some (configurable) limit

Choosing the Language

- We need function names, variables, and constants
- The functions come from
  - a handful of built-in functions
  - functions appearing in the theorem
  - functions appearing in the definitions of functions in the theorem
  - . . . up to some (configurable) limit
- The variable names come from
  - variables appearing in the theorem

Choosing the Language

- We need function names, variables, and constants
- The functions come from
  - a handful of built-in functions
  - functions appearing in the theorem
  - functions appearing in the definitions of functions in the theorem
  - ... up to some (configurable) limit
- The variable names come from
  - variables appearing in the theorem
- The constants come from
  - a handful of built-in constants

Choosing the Language

- We need function names, variables, and constants

Choosing the Language

- We need function names, variables, and constants
- The functions come from
  - a handful of built-in functions
  - functions appearing in the theorem
  - functions appearing in the definitions of functions in the theorem
  - . . . up to some (configurable) limit

Choosing the Language

- We need function names, variables, and constants
- The functions come from
  - a handful of built-in functions
  - functions appearing in the theorem
  - functions appearing in the definitions of functions in the theorem
  - ... up to some (configurable) limit
- The variable names come from
  - variables appearing in the theorem

Choosing the Language

- We need function names, variables, and constants
- The functions come from
  - a handful of built-in functions
  - functions appearing in the theorem
  - functions appearing in the definitions of functions in the theorem
  - . . . up to some (configurable) limit
- The variable names come from
  - variables appearing in the theorem
- The constants come from
  - a handful of built-in constants

Choosing the Handful of Built-in Functions

- Unary functions are better than binary functions
- Forgetting about types is a common source of errors
- Types also come up with theory evolution

Choosing the Handful of Built-in Functions

- Unary functions are better than binary functions
- Forgetting about types is a common source of errors
- Types also come up with theory evolution

- Choose “type” predicates
- As in tau, this means unary boolean predicates

Choosing the Handful of Built-in Functions

- Unary functions are better than binary functions
- Forgetting about types is a common source of errors
- Types also come up with theory evolution

- Choose “type” predicates
- As in tau, this means unary boolean predicates

- Includes `atom`, `integerp`, `true-listp`, ...
- Also compound predicates, e.g.,
  `(and (acl2-numberp x) (not (equal x '0)))`

Generating Function Calls

- The functions expect one or more arguments, which need to be generated
  - Use the variables and constant symbols
  - Use terms built up from functions in the theorem (and their definitions. . .)
  - **Do not** use nested built-in functions

Generating More Complex Boolean Expressions

- Richer hypotheses can be considered by allowing boolean expressions up to a (configurable) depth limit
- Only known predicates are combined this way, e.g., the built-in functions

Generating More Complex Boolean Expressions

- Richer hypotheses can be considered by allowing boolean expressions up to a (configurable) depth limit
- Only known predicates are combined this way, e.g., the built-in functions

- The user can enable exploring comparison operators, like `equal` and `<<`
- Note that the arguments to these are the same as the arguments to other functions,
  - variables and constant symbols
  - terms built up from functions in the theorem (and their definitions. . .)
  - **But not** nested built-in functions
- This leads to many duplicates with the built-in compound predicates, so we disable those when comparisons are enabled.

Not Drowning while Generating Expressions

- Ideally, we will generate enough expressions to find useful suggestions, but not so many to make DrLA glacially slow

Not Drowning while Generating Expressions

- Ideally, we will generate enough expressions to find useful suggestions, but not so many to make DrLA glacially slow

- Do not nest the boolean predicates
- Avoid duplication when using AND/OR/NOT
- Use commutativity and associativity of AND/OR
- Eliminate obvious redundant terms, e.g., `(and X X)`
- Eliminate subtly redundant terms, e.g., `(and X Y)` where `X` implies `Y`

Not Drowning while Generating Expressions

- Ideally, we will generate enough expressions to find useful suggestions, but not so many to make DrLA glacially slow

- Do not nest the boolean predicates
- Avoid duplication when using AND/OR/NOT
- Use commutativity and associativity of AND/OR
- Eliminate obvious redundant terms, e.g., `(and X X)`
- Eliminate subtly redundant terms, e.g., `(and X Y)` where `X` implies `Y`

- Also, generate the expression lazily, so we never need to build an ACL2 list with all expressions

Ranking the Suggestions

- DrLA considers a very large space of possible hypotheses
- Users do not want to see the best 50,000 suggestions

Ranking the Suggestions

- DrLA considers a very large space of possible hypotheses
- Users do not want to see the best 50,000 suggestions

- An approach to ranking the suggestions is to use subsumption
- If both `X` and `Y` are suggested, DrLA will pick only `X` if `X` is more general than `Y`
- E.g., if both `rationalp` and `acl2-numberp` are possible hypotheses, choose `acl2-numberp`

Outline

- Context
- Introducing DrLA
- A Classic Example
- Conclusion

Reverse Reverse

- Beginners are often surprised to find that this is not a theorem in ACL2

```
(reverse (reverse x)) = x
```

- Experienced ACL2 users immediately recognize the missing hypothesis

```
(true-listp x)
```

Generating Terms

The first step is to generate terms, not necessarily booleans

- `(posp x)`
- `(consp x)`
- `(reverse x)`
- `(revappend x 0)`
- `(equal (reverse x) x)`

Generating Terms

The first step is to generate terms, not necessarily booleans

- `(posp x)`
- `(consp x)`
- `(reverse x)`
- `(revappend x 0)`
- `(equal (reverse x) x)`

Note that DrLA will not nest the built-in predicates, so it will not consider terms like

- `(posp (consp x))`

Generating Terms

The first step is to generate terms, not necessarily booleans

- `(posp x)`
- `(consp x)`
- `(reverse x)`
- `(revappend x 0)`
- `(equal (reverse x) x)`

Note that DrLA will not nest the built-in predicates, so it will not consider terms like

- `(posp (consp x))`

DrLA also considers boolean combinations of such expressions

- `(or (posp x) (consp x))`

Finding Useful Suggestions

DrLA will find many, many candidate hypotheses, including

- `(equal x 'nil)`
- `(equal (revappend x x) 'nil)`
- `(true-listp (revappend x x))`
- `(and (consp x) (true-listp x))`
- `(and (true-listp x) (equal (reverse x) 'nil))`
- `(true-listp x)`
- ...

Finding Useful Suggestions

DrLA then uses subsumption to prune the possible suggestions

```
(implies (or (stringp x)
             (true-listp x))
         (equal (reverse (reverse x)) x))
```

Finding Useful Suggestions

DrLA then uses subsumption to prune the possible suggestions

```
(implies (or (stringp x)
             (true-listp x))
         (equal (reverse (reverse x)) x))
```

Surprise! `reverse` works on lists and strings

Outline

- Context
- Introducing DrLA
- A Classic Example
- Conclusion

Future Work

- Take advantage of powerful features of `cgen`
- E.g., use `defattach` to support constrained functions

Future Work

- Take advantage of powerful features of `cgen`
- E.g., use `defattach` to support constrained functions

- Better selection of initial vocabulary
- E.g., let user suggest some functions to use as predicates or general terms
- (Recall that DrLA uses these types of functions differently)
- Users may also provide a list of functions <u>not</u> to consider

Future Work

- Take advantage of powerful features of `cgen`
- E.g., use `defattach` to support constrained functions

- Better selection of initial vocabulary
- E.g., let user suggest some functions to use as predicates or general terms
- (Recall that DrLA uses these types of functions differently)
- Users may also provide a list of functions <u>not</u> to consider

- Use ML to suggest the initial vocabulary
- E.g., checkpoints that look like this benefit from `true-listp`
- Also, use locality to suggest the initial vocabulary
- E.g., I see that you've been using `balanced-p` in many recent theorems

Future Work

- Take advantage of powerful features of `cgen`
- E.g., use `defattach` to support constrained functions

- Better selection of initial vocabulary
- E.g., let user suggest some functions to use as predicates or general terms
- (Recall that DrLA uses these types of functions differently)
- Users may also provide a list of functions <u>not</u> to consider

- Use ML to suggest the initial vocabulary
- E.g., checkpoints that look like this benefit from `true-listp`
- Also, use locality to suggest the initial vocabulary
- E.g., I see that you've been using `balanced-p` in many recent theorems

- Integrate with student front-ends, e.g., ProofPad

Thanks!

False Witnesses

Consider this (non-)theorem

```
(equal (<= (* k x) (* k y))
       (<= x y))
```

The ideal missing hypothesis is `(and (rationalp k) (< 0 k))`

But that's assuming the intended use where all variables are numbers

A possible (false) witness is `k=-1`, `x=NIL`, and `y=NIL`

Another (false) witness is `k=-1`, `x=0`, and `y=0`