Theorems About Programming Languages in ACL2

Sol Swords William Cook
{sswords,wcook}@cs.utexas.edu

Department of Computer Science
University of Texas at Austin

ACL2 Workshop, FLoC 2006

Outline

1. Introduction: Mechanizing Programming Language Metatheory
2. Representing terms, types, and other structures
3. Directing Proofs
4. Further Questions

Proofs about Programming Languages?

What might we want to prove about a programming language?

- Our running example: A well-typed program in the simply-typed λ-calculus with booleans will never get stuck.
- Well-typed programs always terminate.
- The subtyping relation is transitive.
- $\ldots$

Such properties can be surprisingly difficult to prove even when they seem intuitively obvious.

- Proofs are typically complex inductions, either over syntactic forms or sets of rules.
- Most subcases are easy but tedious.
- Adding language features leads to more and more subcases.

Challenges for mechanization

The fact that proofs are large and tedious suggests the use of a theorem prover. Problems to face:

- Most proofs have a few hard parts that require significant ingenuity, requiring user interaction.
- Some ideas are easy to gloss over in hand proofs but difficult to formalize.
  - Example: variable binding. In hand proofs, it is assumed that all variable names are unique; this is difficult to make explicit for a mechanized proof.
- It is difficult to define complex recursive data structures so that they can be reasoned about smoothly in ACL2.

Outline

Representing Terms

- ACL2 users work with simple recursive data structures all the time: lists, trees.
- Expressions, types, and other syntactic forms must be represented by more complex structures.
- It is painful to prove theorems about such structures if the underlying cons representation is exposed.
- Best method: Encapsulate operations in functions and prove sufficient theorems about them so that they can be disabled, thus hiding the cons representation.

We introduce the macro `defsum` which automates these definitions and proofs.

Example: Types in the λ-calculus

Syntax of types:

$$\begin{array}{rcll} T & ::= & \text{Bool} & \textit{Boolean} \\ & & \mid\ T \rightarrow T & \textit{Function} \end{array}$$

English:
A type is either the constant Bool or a function type composed of a domain and range which are both types.

Defsum form for ACL2:

```
(defsum stype
  (BOOL)
  (FUN (stype-p domain) (stype-p range)))
```

Functions defined by defsum

```
(defsum stype
  (BOOL)
  (FUN (stype-p domain) (stype-p range)))
```

Introduces the following functions:

- Sum recognizer: `stype-p`
- Product recognizers: `bool-p`, `fun-p`
- Constructors: `bool`, `fun`
- Destructors: `fun-domain`, `fun-range`
- Measure: `stype-measure`

Total: 8 functions.

Theorems proved by defsum

Defsum automatically proves enough theorems about these functions to allow reasoning about these types without reference to the underlying cons structure. Examples:

- `(fun-p (fun domain range))`
- ```
  (implies (and (stype-p domain)
                (stype-p range))
           (stype-p (fun domain range)))
  ```
- ```
  (equal (fun-domain (fun domain range))
         domain)
  ```
- `(not (equal (fun domain range) range))`
- ```
  (implies
   (not (equal domain (fun-domain x)))
   (not (equal (fun domain range) x)))
  ```

Total: 35 theorems.
  ```

Pattern-match

Pattern-match is a companion macro to defsum allowing ML or Haskell-style pattern-matching over sum types. Example:

```
(defun print-stype (x)
  (pattern-match x
    ((BOOL)    (cw "bool"))
    ((FUN a b) (cw "(~x0) -> (~x1)"
                   (print-stype a)
                   (print-stype b)))))
```

Pattern-match

Equivalent without pattern-match:

```
(defun print-stype (x)
  (if (bool-p x)
      (cw "bool")
    (if (fun-p x)
        (let ((a (fun-domain x))
              (b (fun-range x)))
           (cw "(~x0) -> (~x1)"
               (print-stype a)
               (print-stype b)))
      nil)))
```

Each defsum form introduces macros which enable pattern-match to recognize the newly defined constructors.

Outline

Theorem example: Preservation

Theorem: Preservation. If an expression x has type T under typing context Γ, and x evaluates to x', then x' also has type T under Γ.

Notes:

- A typing context is an alist mapping free variables to their assumed types. At top level, we expect there to be no free variables and we work with the empty context.
- We haven't shown how to determine whether an expression has a type.
- We haven't shown how to determine whether one expression evaluates to another.

Evaluation and Typing as Functions

We can define evaluation and typing as functions $\text{Type}(x, \Gamma)$ and $\text{Eval}(x)$:

Theorem: Preservation. If $\text{Type}(x, \Gamma) = T$ and $\text{Eval}(x) = x'$, then $\text{Type}(x', \Gamma) = T$.

Problems:

- Type and Eval are not functions in every language. Examples:
  - Eval is not a function in nondeterministic languages.
  - Type is not a function in languages with subtyping.
- Reasoning about Type and Eval as functions requires leading the theorem prover by using lots of hints: frustrating and hard to debug.

For future reference, call this the “direct method” of proving these theorems.

Evaluation and Typing Derivations

Alternative: Define evaluation and typing relations $x \rightsquigarrow x'$ and $\Gamma \vdash x : T$ in terms of the existence of a derivation: an object which shows which rules are applied when in order to prove that the relation holds.

Theorem: Preservation. Given derivations of $\Gamma \vdash x : T$ and $x \rightsquigarrow x'$, one can construct a derivation of $\Gamma \vdash x' : T$.

Proof is simple: Define a function that constructs the derivation of $\Gamma \vdash x' : T$. This function provides the induction scheme and the structure of the final proof, which is to verify that this construction is correct if the input derivations are correct.

Direct Method versus Derivation Method

Trade-offs:

- The direct method saves work on the many simple, obvious, uninteresting cases, but is hard to drive through the difficult parts.
- Derivation functions provide a direct way of guiding the prover through difficult cases, but must explicitly specify the simple cases as well.
- **The direct method makes better use of built-in ACL2 heuristics, whereas derivation functions give more control over the prover.**

We completed the proof of the soundness of the simply-typed λ-calculus using the derivation method; only `:induct` and `:in-theory` hints were used.

Outline

Questions

- Is there a method with tight control over the prover **and** heuristics that plow through the easy parts?
- Can these methods work on larger problems?
  - Example: The POPLMark Challenge - prove the transitivity of the subtyping relation in the language $F_{<:}$.
- Best way to reason about variable bindings?
  - Very problematic in more complex languages, like $F_{<:}$.
  - The current most successful method, Higher Order Abstract Syntax, is unavailable to ACL2 and other first-order logics.
  - Congruences over α-equivalence helpful?