The ACL2s Language

Pete Manolios
Northeastern

Logic and Computation, 1/17/2019

Let

- ```
  (let ((v1 x1)
        ...
        (vn xn))
    body)
  ```

binds its local variables, the `vi`, in parallel, to the values of the `xi`, and evaluates body using that binding

- For example:

```
(let ((x '(a b c))
      (y '(c d)))
  (app (app x y) (app x y)))
```

- evaluates to `(a b c c d a b c c d)`
- This saves us having to type `'(a b c)` and `'(c d)` multiple times
- Notice how the use of quotes: instead of `(list 'a 'b 'c)` we have `'(a b c)`
```

Let*

- Maybe we can avoid having to type `(app x y)` multiple times. What about?

```
(let ((x '(a b c))
      (y '(c d))
      (z (app x y)))
  (app z z))
```

```
(let ((x '(a b c))
      (y '(c d)))
  (app (app x y) (app x y)))
```

- This does not work. Why not? Because let binds in parallel, so x and y in the z binding are not yet bound

Let*

- What we really want is a binding form that binds sequentially. That is what `let*` does.

```
(let* ((v1 x1)
       ...
       (vn xn))
  body)
```

  binds its local variables, the `vi`, sequentially, to the values of the `xi`, and evaluates body using that binding, so this works:

```
(let* ((x '(a b c))
       (y '(c d))
       (z (app x y)))
  (app z z))
```

```
(let ((x '(a b c))
      (y '(c d)))
  (app (app x y) (app x y)))
```

- `let` and `let*` give us **abbreviation power** and **efficiency**
- Why might `let` be preferable to `let*`?

Data Definitions

- Demo
- Singleton types
- Recognizers
- Enumerated Types
- Range Types
- Product Types
- Records
- Constructors & Accessors
- Listof Combinator
- Union Types
- Recursive Types
- Data-driven Function Definitions
- Mutually Recursive Data Types

Slides by Pete Manolios for CS2800, Logic & Computation, NU 2019