Lecture 2

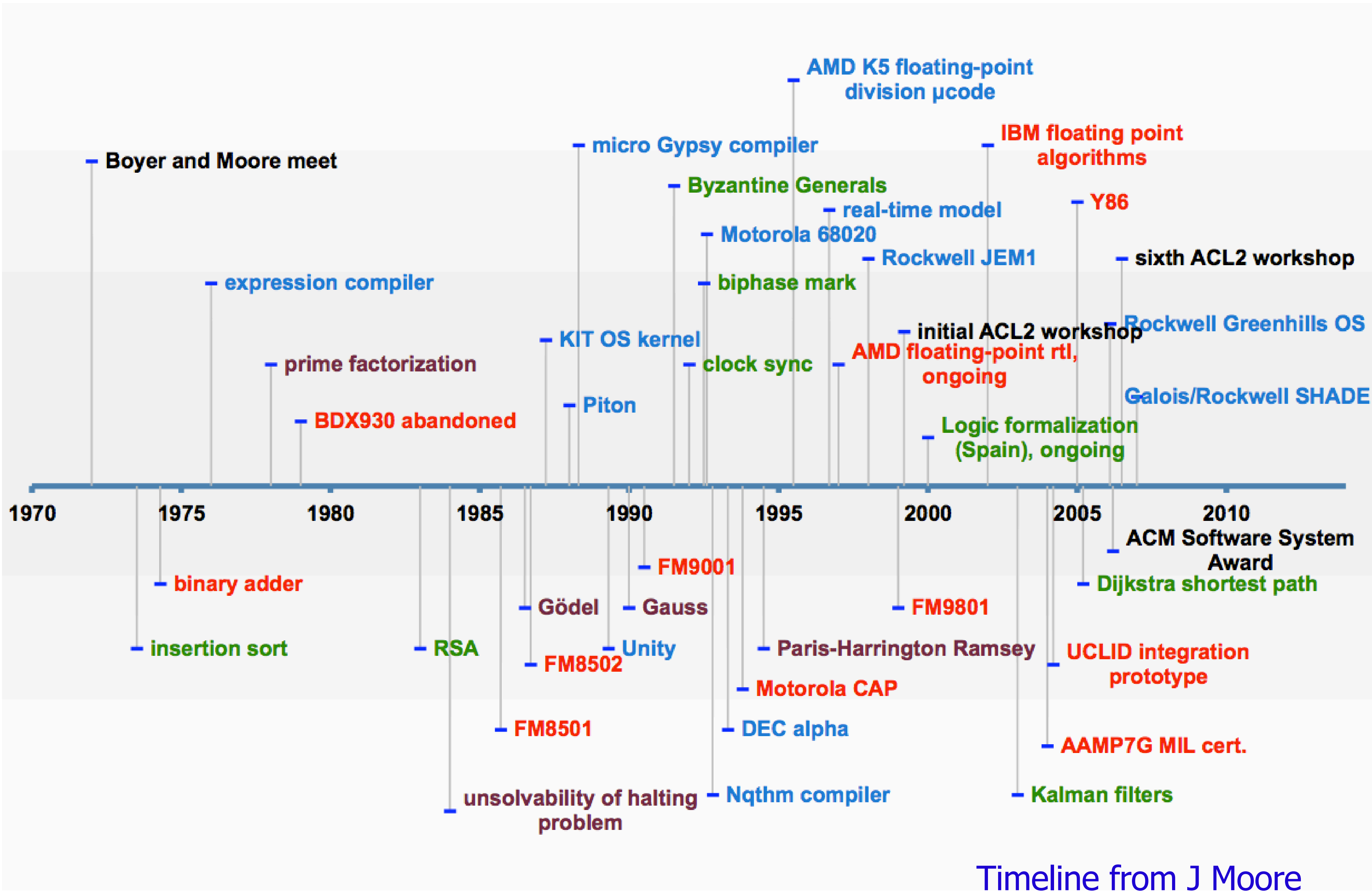
Pete Manolios
Northeastern

Boyer-Moore Theorem Provers

- **1970's**
  - Edingurgh Pure Lisp Theorem Prover (1973)
  - A Computational Logic (1978)
- **1980's**
  - NQTHM (1981)
  - ACL2 (1989) A Computational Logic for Applicative Common Lisp
- **1990's-Present**
  - Kaufmman joins as developer
  - Workshops (10 already); huge regression suite
- **2000's:**
  - ACL2 books
  - Development environments (ACL2 Sedan)
  - 2005 ACM Software System Award (Boyer, Kaufmann, Moore)

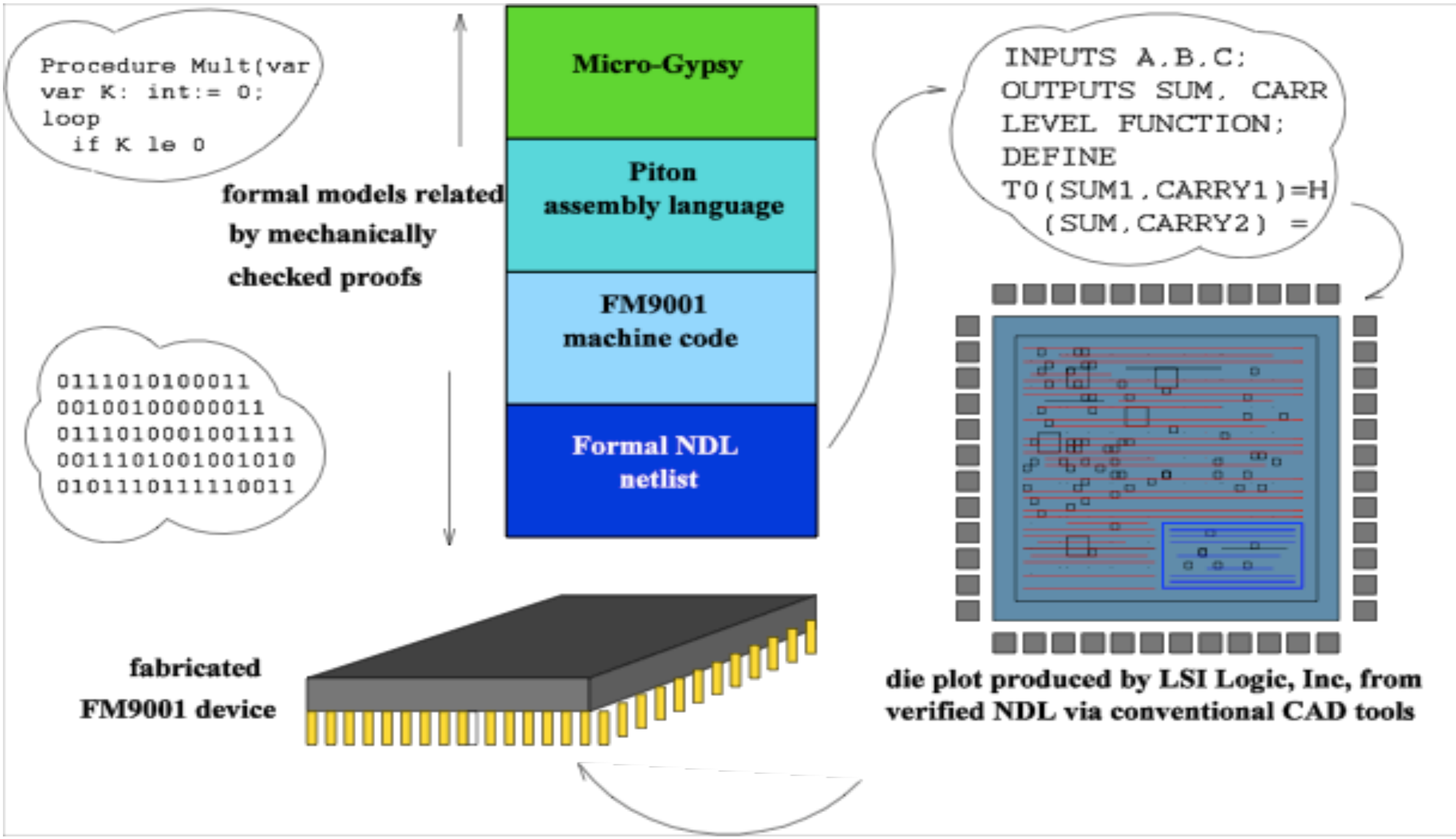
Boyer-Moore Theorems Proved

- **1970's: Simple List Processing**
  - Associativity of append
  - Prime factorizations are unique
- **1980's: Academic Math & CS**
  - Invertibility of RSA
  - Undecidability of halting problem
  - Gödel's First Incompleteness Theorem
  - Gauss' Law of Quadratic Reciprocity
  - CLI Stack:
    - Microprocessor
    - Assembler-linker-loader, Compiler, OS
    - High-level language



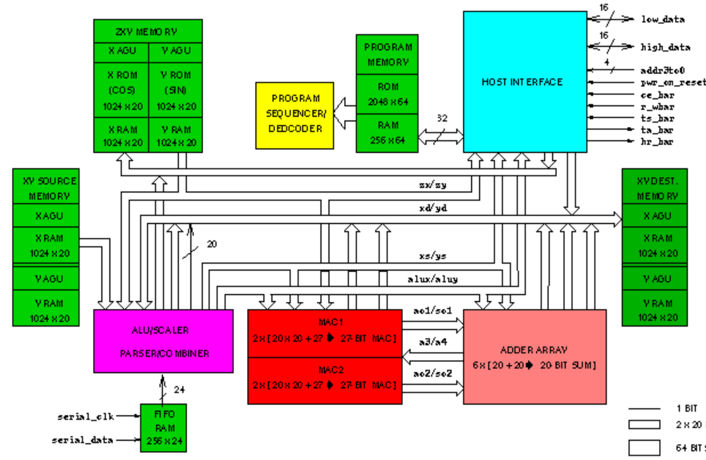
Slides by Pete Manolios for CS4820

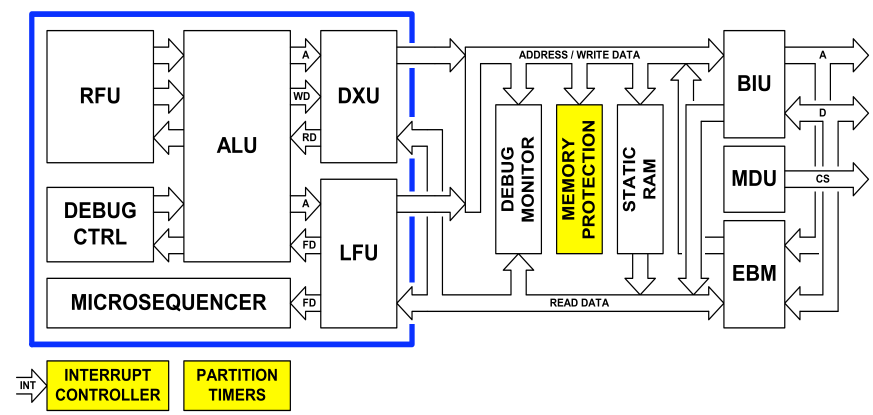
CLI Stack


Procedure Mult(var
var K: int:= 0;
loop
if K le 0
formal models related
by mechanically
checked proofs
0111010100011
0010010000011
0111010001001111
0011101001001010
0101110111110011
Micro-Gypsy
Piton
assembly language
FM9001
machine code
Formal NDL
netlist
INPUTS A,B,C;
OUTPUTS SUM, CARR
LEVEL FUNCTION;
DEFINE
T0(SUM1,CARRY1)=H
(SUM,CARRY2) =
fabricated
FM9001 device
die plot produced by LSI Logic, Inc, from
verified NDL via conventional CAD tools

Industrial Applications

- FDIV AMD Floating Point, IBM …
- Motorola CAP DSP
  - Bit/cycle-accurate model
  - Run fasters than SPW model
  - Proved correctness of pipeline hazard detection in microcode
  - Verified microcode programs
- Rockwell Collins JEM1
- Rockwell Collins AAMP7
  - MILS EAL-7 certification from NSA for their crypto processor
  - Verified separation kernel
- Centaur: Media Unit





So What?

Mechanized reasoning for commercial systems ✔

- Scalability to industrial problems
- Tool maturity
- Human talent
- Repeatability
- Time to market
- ROI vs other methods

What's Next?

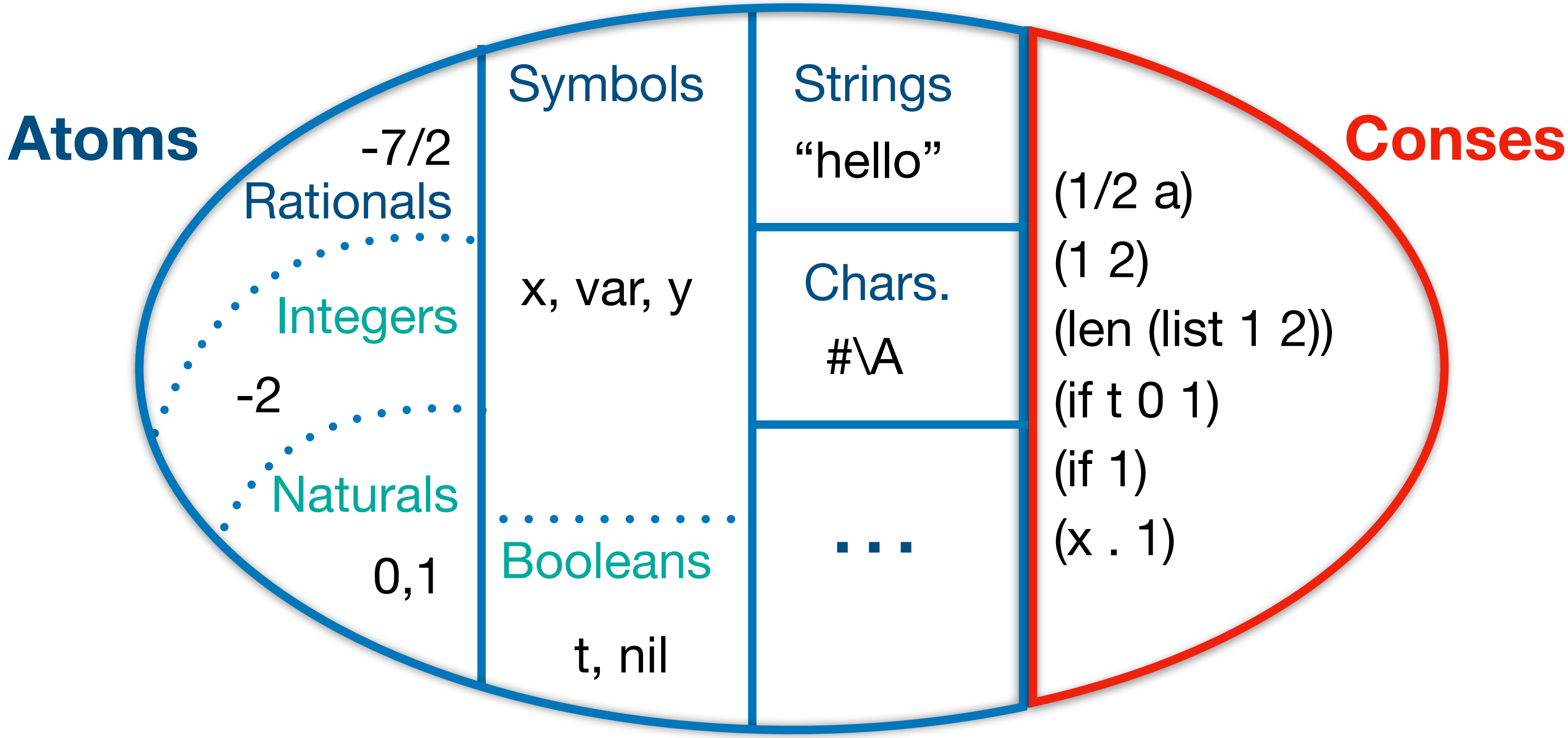
Computer-aided reasoning for the masses
Teach freshmen how to reason about programs

ACL2 is ...

- **A programming language:**
  - Applicative, functional subset of Lisp
  - Compilable and executable
  - Untyped, first-order
- **A mathematical logic:**
  - First-order predicate calculus
  - With equality, induction, recursive definitions
  - Ordinals up to ε_0 (termination & induction)

ACL2 Universe

All = Conses ∪ Atoms



Lists = Conses ∪ { ( ) }

True-lists = $\bigcup_{i \in \mathbb{N}} L_i$

$L_0 = \{\, () \,\}$, $L_{i+1} = L_i \cup \{(\texttt{cons x l}) : \texttt{x} \in \texttt{All},\ \texttt{l} \in L_i\}$

Slides by Pete Manolios for CS4820

Data Definitions

- Allow us to write recognizers & enumerators for subsets of the universe
- Singleton types
- Recognizers
- Enumerated Types
- Range Types
- Product Types
- Records
- Constructors & Accessors
- Listof Combinator
- Union Types
- Recursive Types
- Mutually Recursive Data Types
- Custom types

Slides by Pete Manolios for CS4820

DEMO

Expressions

- "Expressions" (or "terms") are elements of a subset of U (the Universe)
- Evaluation maps expressions to ACL2 objects
- ⟦*expr*⟧ denotes the semantics of *expr*
  - or what *expr* evaluates to at the REPL
- Constants are expressions that evaluate to themselves
  - `⟦t⟧ = t`
  - `⟦nil⟧ = nil`
  - `⟦6⟧ = 6`
  - `⟦-21⟧ = -21`

Lazy vs Strict

- Semantics of `if`
  - ⟦`(if` *test then else*`)`⟧ = ⟦*then*⟧ , when ⟦*test*⟧ ≠ `nil` (Generalized Booleans)
  - ⟦`(if` *test then else*`)`⟧ = ⟦*else*⟧ , when ⟦*test*⟧ = `nil`
- `if` is lazy:
  - first ACL2s evaluates *test*, i.e., it computes ⟦*test*⟧
  - if ⟦*test*⟧ ≠ `nil` then ACL2s returns ⟦*then*⟧
  - otherwise, it returns ⟦*else*⟧
- So, *test* is always evaluated, but only one of *then*, *else* is
- All other functions are strict
  - ACL2s evaluates all of the arguments to the function
  - Then ACL2s applies the function to evaluated results

Function Definitions

- Why does this definition make sense?
- Because it terminates
- A key idea every time you define a program is to convince yourself that on every recursive call, some parameter decreases in a well-founded way
- Hmm, can lists be circular? then what?
- Lists are non-circular in ACL2s, which is why this works
- Termination is one of the key ideas in CS
- Note that data driven definitions always terminate

```
(defunc mlen (l)
  :input-contract (true-listp l)
  :output-contract (natp (mlen l))
  (if (endp l)
    0
    (+ 1 (mlen (rest l)))))
```

```
(defunc mlen (l)
  :input-contract (true-listp l)
  :output-contract (natp (mlen l))
  (if (endp l)
    (+ 1 (mlen (rest l)))
    0))
```

What if I wrote this?

Invariants

- On to another key concept: invariants
- What is an invariant?
  - A property that is always satisfied in all executions of a program is an invariant
  - Properties are associated with program locations
- For example let `I` = `(true-listp l)`
- Then `I` is an invariant because at that location in the program it always holds
- Why?
- The input contract requires it

```
(defunc mlen (l)
  :input-contract (true-listp l)
  :output-contract (natp (mlen l))
  (if (endp l)
    0
    (+ 1 (mlen (rest l)))))
```

```
(defunc mlen (l)
  :input-contract (true-listp l)
  :output-contract (natp (mlen l))
  (if (endp l)
    0
    (+ 1 (mlen (rest {I}l)))))
```

Contracts

- A simple, useful class of invariants that you should always check are contracts
- Every function has an input contract
- For every function call, we must be able to
  - statically establish that the input contract of the function is satisfied
- In ACL2s we can specify contracts
  - ACL2s checks them for us

```
(defunc mlen (l)
  :input-contract (true-listp l)
  :output-contract (natp (mlen l))
  (if (endp l)
    0
    (+ 1 (mlen (rest l)))))
```

All elite programmers I know think in terms of invariants

Contracts

- Body contracts
  - `1. endp: (listp l)`
  - `2. rest: (listp l)`
  - `3. mlen: (true-listp l)`
  - `4. +: (acl2-numberp 1)`
    `(acl2-numberp (mlen (rest l)))`
  - `5. if: t`
- Function contract
  - `(true-listp l) => (natp (mlen l))`
- Contract contracts
  - 6. `true-listp: t` (`true-listp` is a recognizer)
  - 7. `mlen: (true-listp l)` (input contract!)
  - 8. `natp: t` (`natp` is a recognizer)

```
(defunc mlen (l)
  :input-contract (true-listp l)
  :output-contract (natp (mlen l))
  (if (endp l)
    0
    (+ 1 (mlen (rest l)))))
```

```
(defunc mlen (l)
  :input-contract {6}(true-listp l)
  :output-contract {8}(natp {7}(mlen l))
  {5}(if {1}(endp l)
    0
    {4}(+ 1 {3}(mlen {2}(rest l)))))
```

- Every time you write a program, (not just for for this class), check body and function contracts!
- You can think of invariants as assertions
  - {i} means that every time program execution reaches this point then {i} is true

Static Checking

- Body contracts
  - `1. endp: (listp l)`
  - `2. rest: (listp l)`
  - `3. mlen: (true-listp l)`
  - `4. +: (acl2-numberp 1)`

    `(acl2-numberp (mlen (rest l)))`
  - `5. if: t`
- Function contract, contract contracts …
- Static checking of contracts
  - Before the definition is accepted we prove all the contracts
  - During execution, only top-level input contracts are checked
  - We have assurance that, at the language level, code will run without any runtime errors
- Static checking of contracts is hard, which is why it is not supported in most PLs

```
(defunc mlen (l)
  :input-contract {6}(listp l)
  :output-contract {8}(natp {7}(mlen l))
  {5}(if {1}(endp l)
    0
    {4}(+ 1 {3}(mlen {2}(rest l)))))
```

Dynamic Checking

- Body contracts
  - `1. endp: (listp l)`
  - `2. rest: (listp l)`
  - `3. mlen: (true-listp l)`
  - `4. +: (acl2-numberp 1)`

    `(acl2-numberp (mlen (rest l)))`
  - `5. if: t`
- Function contract, contract contracts …
- Dynamic checking of contracts
  - We generate code to check the contracts at run-time
  - This code can incur a significant performance penalty
  - Contract violations are possible and will lead to an exception
- Dynamic checking is supported via mechanisms such as assertions; typically used only in development

```
(defunc mlen (l)
  :input-contract {6}(listp l)
  :output-contract {8}(natp {7}(mlen l))
  {5}(if {1}(endp l)
    0
    {4}(+ 1 {3}(mlen {2}(rest l)))))
```

Definitional Principle

- The definition

```
(defunc f (x1 . . . xn)
  :input-contract ic
  :output-contract oc
  body)
```

is admissible provided:

  - f is a new function symbol
  - the xi are distinct variable symbols
  - body is a term, possibly using f recursively as a function symbol, mentioning no variables freely other than the xi
  - the function is terminating
  - ic ⇒ oc is a theorem
  - the body contracts hold under the assumption that ic holds

Definitional Axioms

- When we admit a function, we get the following axiom and theorem
  - `ic ⇒ (f x1 ... xn) = body` (Definitional axiom)
  - `ic ⇒ oc` (Contract theorem)
- In proofs we will not explicitly mention input contracts when using a function definition because contract completion (test?!)
- Why termination? `(f x) = 1 + (f x)` leads to inconsistency
- Why no free vars? `(f x) = y` leads to inconsistency

Next Time

- Measure Functions
- Reasoning about Programs
- Axioms
- Equational Reasoning
- Induction
- Lemmas
- Generalization