# Verified Software-Based Fault Isolation

Greg Morrisett    Gang Tan (Lehigh)
Joe Tassarotti    Edward Gan    Jean-Baptiste Tristan (Oracle Labs)

HARVARD
School of Engineering
and Applied Sciences

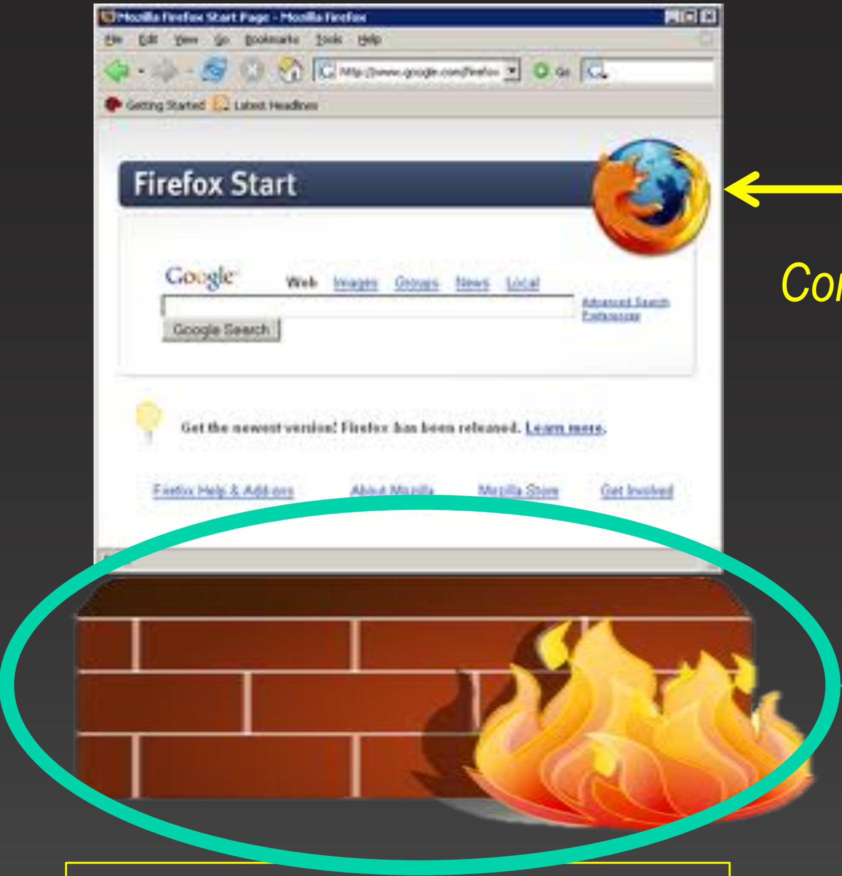
# Extensible Systems

*Combination of HTML and Javascript*

Gmail by Google

Underlying Machine

# Extensible Systems



Get the newest version! Firefox has been released. Learn more.

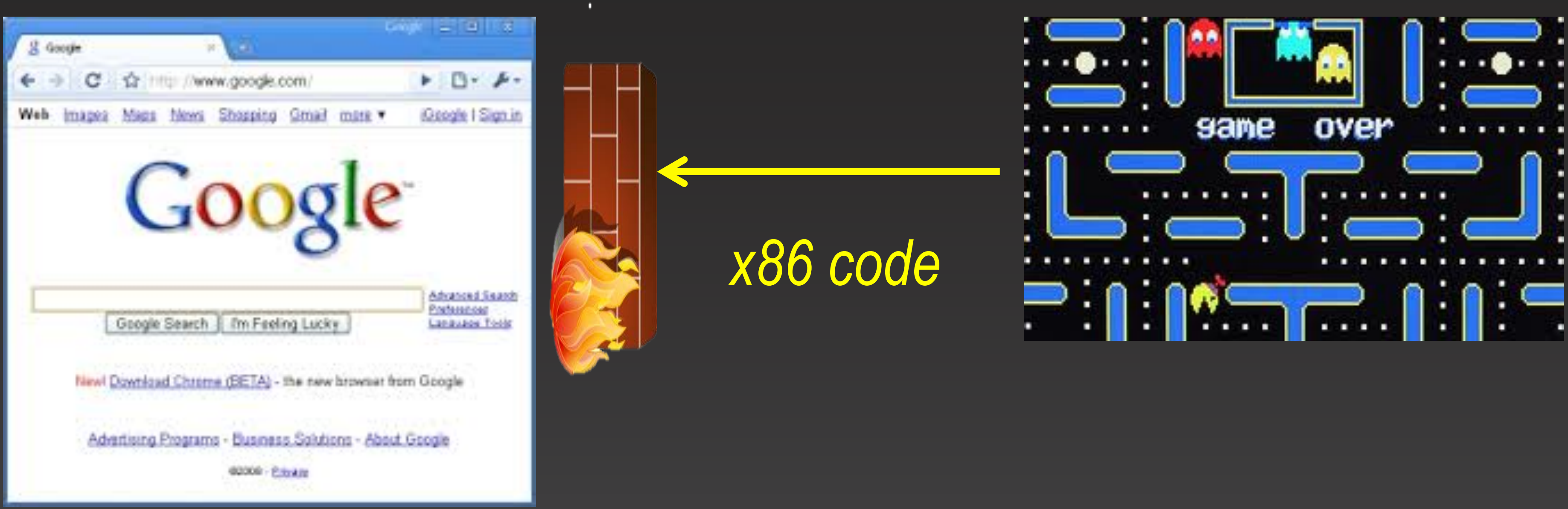
Gmail by Google

*Combination of HTML and Javascript*

An interpreter (or compiler) serving as a reference monitor, that imposes a sandbox policy on the code.

Underlying Machine

# Google’s Native Client



x86 code

Underlying Machine

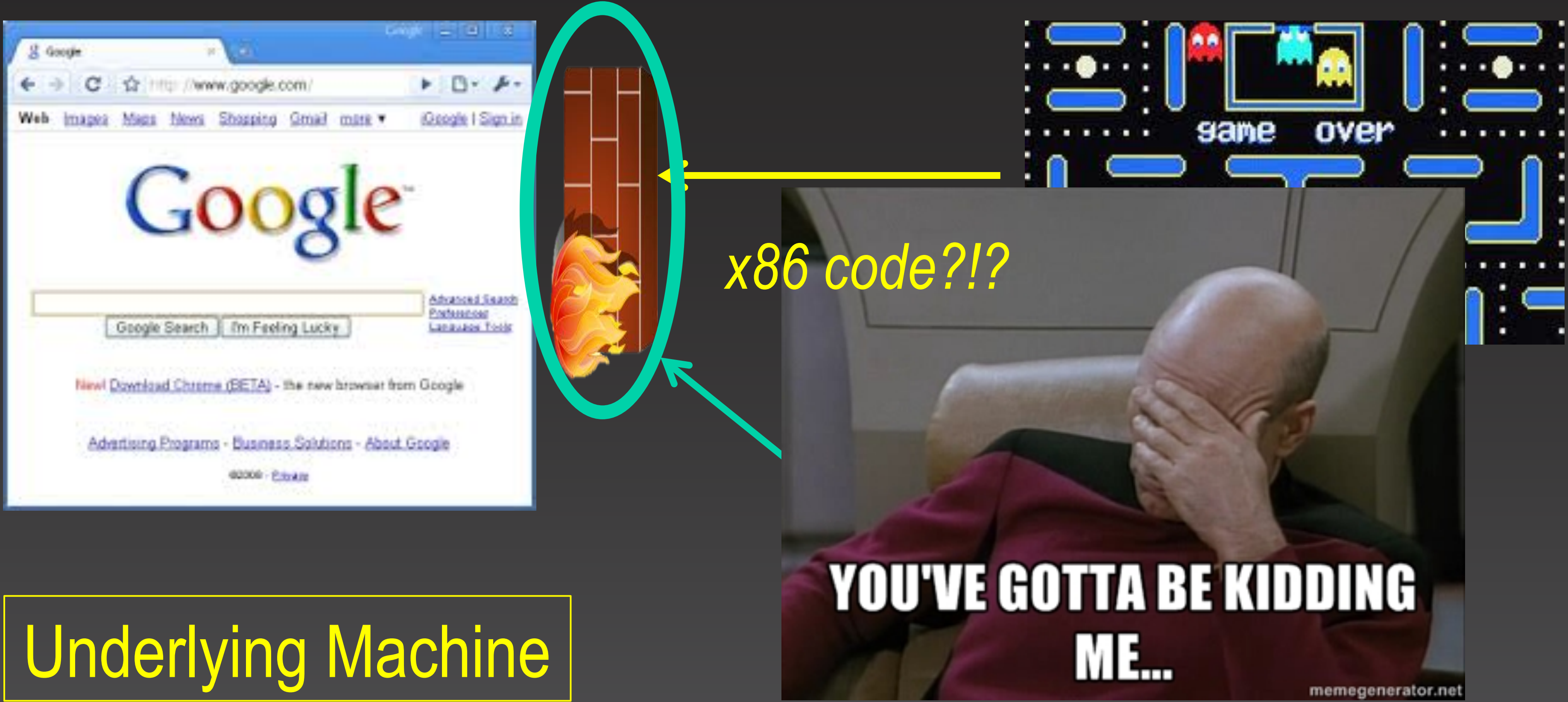
# Google's Native Client

*x86 code*

Supposed to check that the code will stay in a sandbox

Underlying Machine

# Google's Native Client



x86 code?!?

Underlying Machine

# *Software Fault Isolation (SFI)*

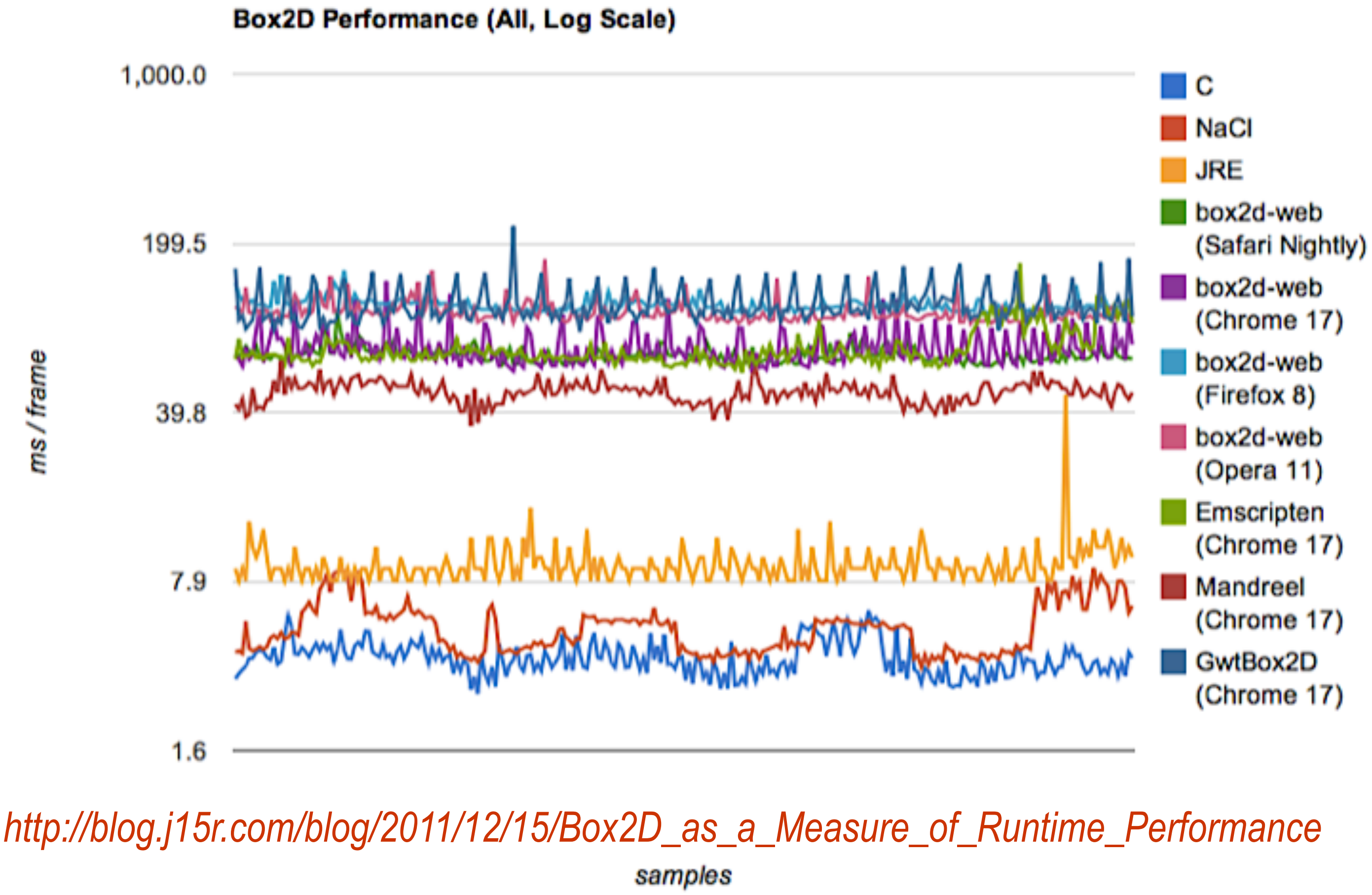
Many applications:

- native code plug-ins for browsers (e.g., Google's NaCl)
- stored procedures in DB (e.g., Wahbe et al.)
- in-kernel device drivers (e.g., Nooks)
- isolating native code for run-times (e.g., Robusta)

would like to have a basic *sandbox* integrity policy.

- all jumps are constrained to a segment of memory
- all writes are constrained to a (separate) segment
- [optionally, all reads are constrained]
- all system (or library) calls are mediated

# Comparing Sandboxes



*http://blog.j15r.com/blog/2011/12/15/Box2D_as_a_Measure_of_Runtime_Performance*

# *Original SFI Approach*

- Wahbe et al. (1994)
- Rewrite MIPS assembly code so that it respects sandbox policy when executed.
  - mask high bits of all effective addresses so they are forced to be in the proper segment.
  - `Mem[A] := r` → `t := mask(A); Mem[t] := r`
- But, code might jump over masking operation.
  - we need the masking and deference to be “atomic”

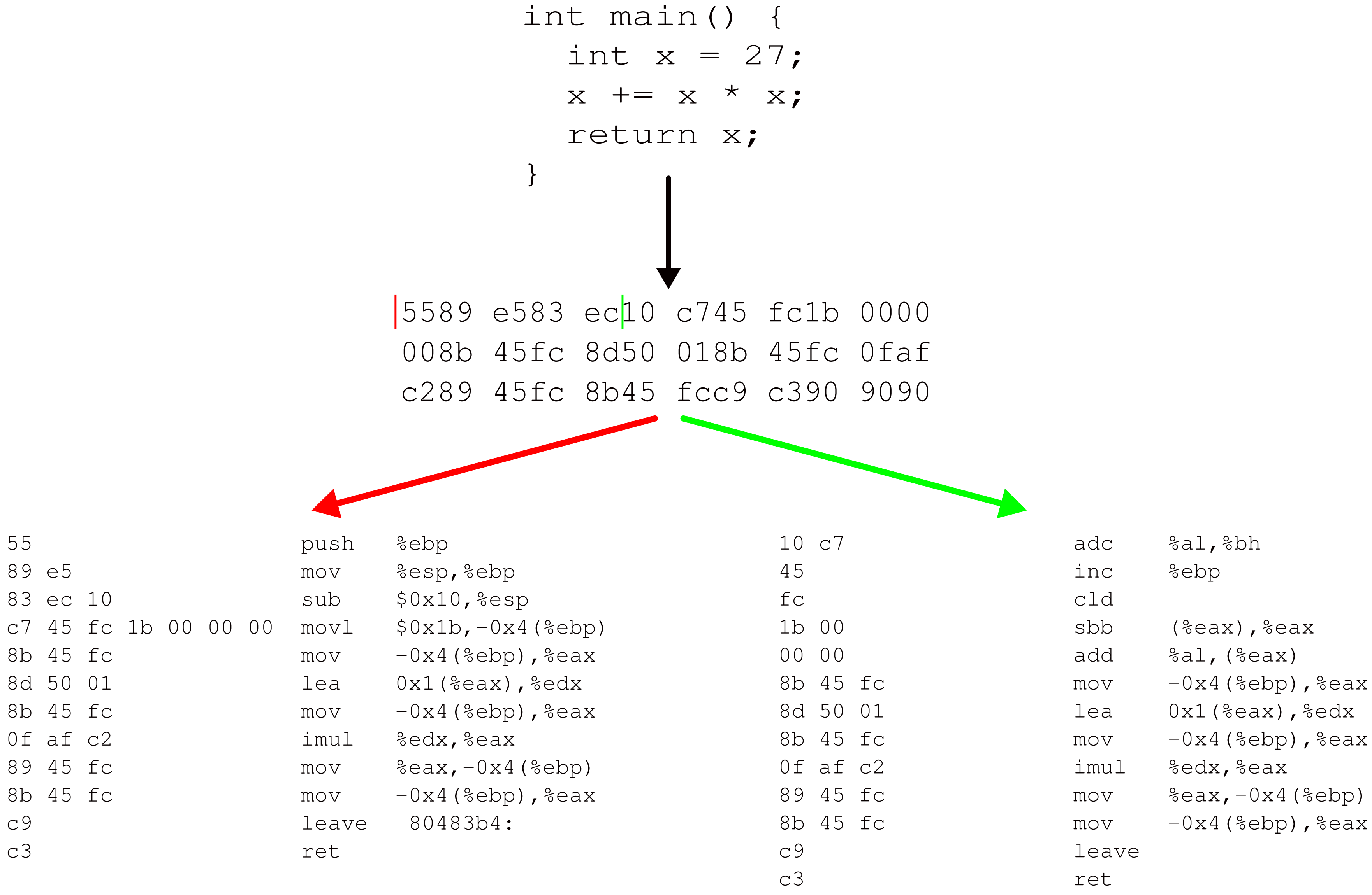
# Berkeley Solution

- Dedicate two registers: 1 for data (D), 1 for control (C).
- Invariant: dedicated registers always point into the proper segment.
  - to store `r` at address `A`: `D := dmask(A); Mem[D] := r`
  - to jump to address `A`: `C := cmask(A); goto C`
- If an attacker jumps over masking operations, the code still stays in the sandbox.

# *What about x86?*

- Can’t afford to burn 2 registers.
  - need some other way to ensure atomicity of checks and uses.
- New problem:  Variable length instructions.
  - there are *multiple* parses we must consider.

# Example



```
int main() {
  int x = 27;
  x += x * x;
  return x;
}
```

```
5589 e583 ec10 c745 fc1b 0000
008b 45fc 8d50 018b 45fc 0faf
c289 45fc 8b45 fcc9 c390 9090
```

```
55                    push   %ebp
89 e5                 mov    %esp,%ebp
83 ec 10              sub    $0x10,%esp
c7 45 fc 1b 00 00 00  movl   $0x1b,-0x4(%ebp)
8b 45 fc              mov    -0x4(%ebp),%eax
8d 50 01              lea    0x1(%eax),%edx
8b 45 fc              mov    -0x4(%ebp),%eax
0f af c2              imul   %edx,%eax
89 45 fc              mov    %eax,-0x4(%ebp)
8b 45 fc              mov    -0x4(%ebp),%eax
c9                    leave   80483b4:
c3                    ret
```

```
10 c7                 adc    %al,%bh
45                    inc    %ebp
fc                    cld
1b 00                 sbb    (%eax),%eax
00 00                 add    %al,(%eax)
8b 45 fc              mov    -0x4(%ebp),%eax
8d 50 01              lea    0x1(%eax),%edx
8b 45 fc              mov    -0x4(%ebp),%eax
0f af c2              imul   %edx,%eax
89 45 fc              mov    %eax,-0x4(%ebp)
8b 45 fc              mov    -0x4(%ebp),%eax
c9                    leave
c3                    ret
```

# *SFI for CISC machines*

- McCamant & Morrisett (2006)
  - force a single parse of the code.
- All direct jumps must be to the beginning of an “atomic” instruction sequence in our parse.
- For computed jumps:
  - don’t allow atomic instruction sequences in our parse to cross a k-byte boundary
  - insert no-ops until we are on a k-byte aligned boundary
  - mask the destination address so it is k-byte aligned
- Overhead: ~20%
  - on 32-bit machines, we can use the segment regs. to cut this to ~5%

# *Google's Native Client (NaCl)*

- Yee et al. (2009)
- New SFI service in Chrome browser.
    - load and run x86 executable
- Modified GCC tool-chain
    - inserts appropriate masking, alignment
- Pepper API
    - access to the browser, DOM, 3D acceleration, etc.
- A checker that ensures code respects the sandbox policy.

# *The Checker*

- A bug in the checker could result in a security breach.
  - earlier implementations of SFI had bugs
  - Google ran a contest and participants found bugs
- Our goal:
  **Prove the correctness of the NaCl checker.**

# The Checker

- A bug in the checker could result in a security breach.
  - earlier implementations of SFI had bugs
  - Google ran a contest and participants found bugs

- Our goal:
  **~~Prove the correctness of the NaCl checker.~~**
  **(too hard)**

# *The Checker*

- A bug in the checker could result in a security breach.
  - earlier implementations of SFI had bugs
  - Google ran a contest and participants found bugs
- Our goal:
  Write and **prove the correctness of a *new* NaCl checker.**

# *Punchline*

We built a new checker for (32-bit) NaCl that we call RockSalt.

- smaller:  80 lines of C
  - based on an idea from a Google Engineer
  - basically a driver operating over automatically generated tables
- faster:  on 200Kloc of compiled C code
  - Google's:  0.9s vs. RockSalt:  0.2s
- stronger: (mostly) proven correct
  - table generation proven correct
  - ML driver proven correct, but manually translated to C

# How RockSalt works

- Specify regexps for parsing legal x86 instructions
  - preclude instructions that might change or override the segment registers.
  - preclude doing a computed jump without first masking the effective address of the destination.

- Compile regexps to a table-based DFA
  - interpret DFA tables &
  - record start positions of instructions &
  - check jump and alignment constraints

- All of this is proven correct.

# *What we proved…*

- If we give the checker a string of bytes B, and the checker accepts, then if we load B into an appropriate x86 context and begin execution, the code will respect the sandbox policy as it runs.
- The real challenge is building a model of the x86.
  - And to gain some confidence that it is correct!
  - We have modeled about 300 different instructions
    - including all the addressing modes, and all of the prefixes.

# *We’re not the first of course…*

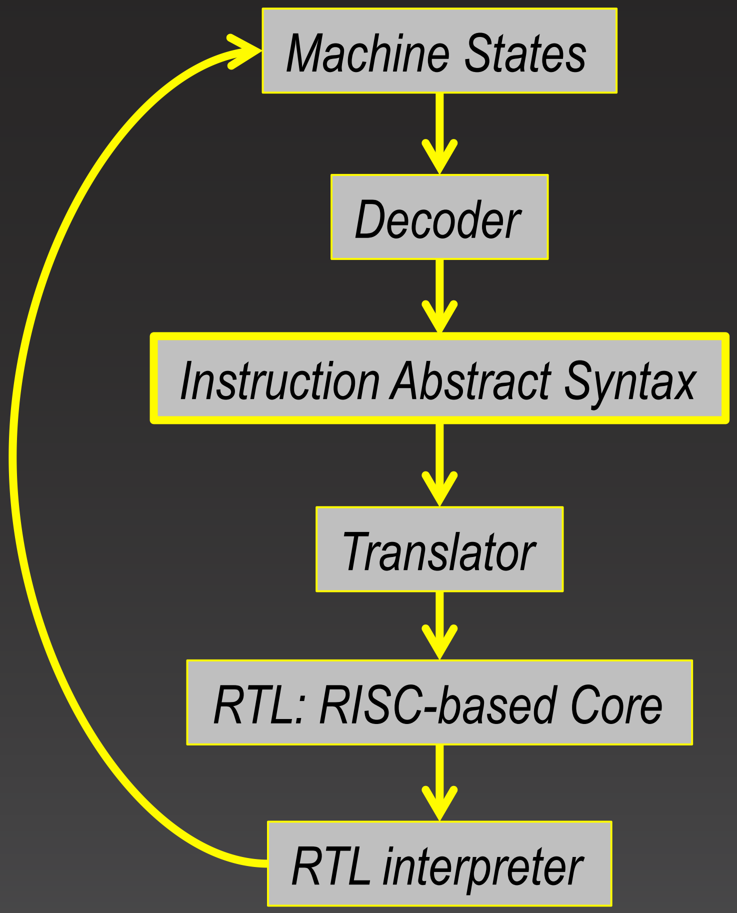
- CompCert’s x86 model (Coq)
  - actually an abstract machine with a notion of stack
  - code is not explicitly represented as bits
- Y86 model (ACL2)
  - tens of instructions, monolothic interpreter
  - but you can extract relatively efficient code for testing!
- Cambridge x86 work (HOL)
  - inspired much of our design
  - their focus was on modeling concurrency (TSO)
  - semantics encoded with predicates (need symbolic computation)

# Our x86 Model

Re-usable, embedded domain-specific languages to specify the semantics.

- Decoder:
  - type-indexed parsing combinators for regular grammars
  - easy denotational semantics
  - operational semantics via *derivatives*
  - proof of adequacy/soundness
- Execution:
  - register transfer language (think GCC)
  - translate x86 instructions into RTLs
  - give operational semantics for RTLs

# Our x86 Model in Coq



Machine States

Decoder

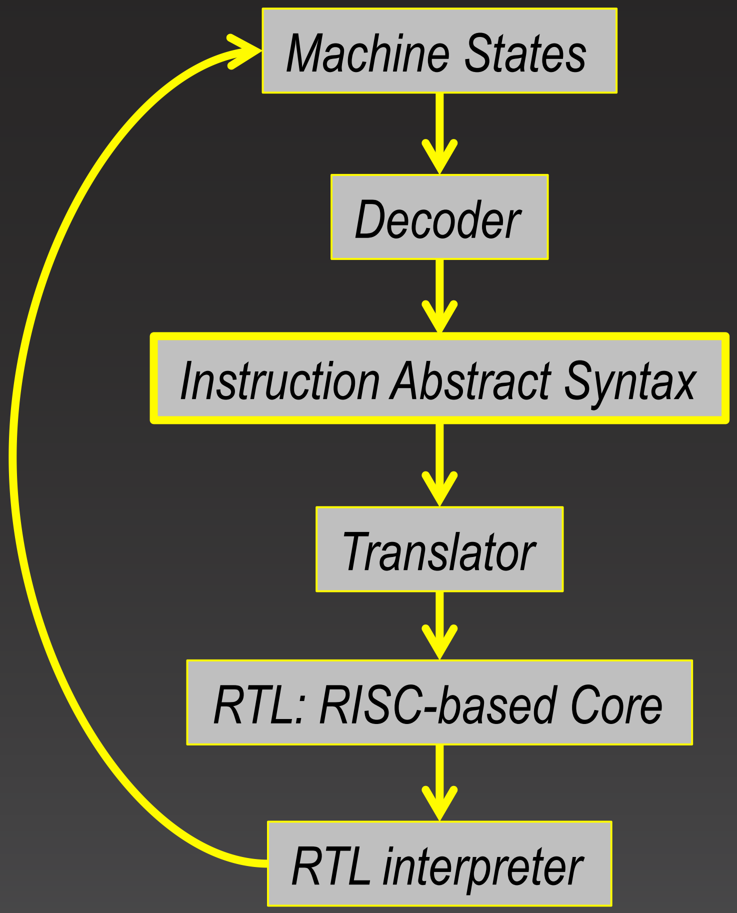
Instruction Abstract Syntax

Translator

RTL: RISC-based Core

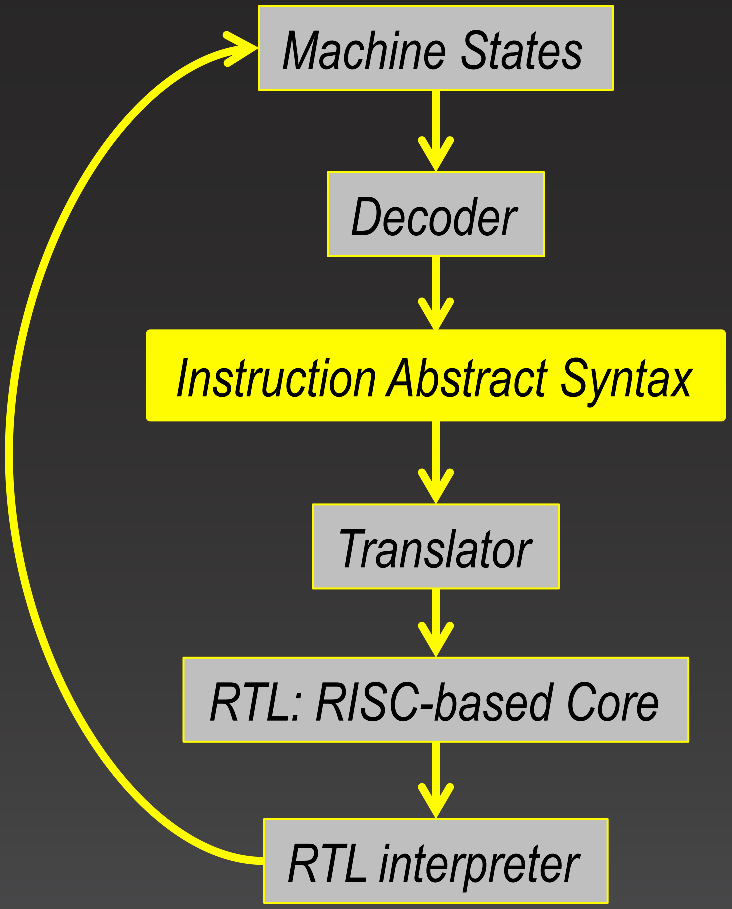
RTL interpreter

# Our x86 Model in Coq



Importantly, we can extract an executable Ocaml interpreter that we can use for validation.

# Our x86 Model in Coq


Machine States

Decoder

Instruction Abstract Syntax

Translator

RTL: RISC-based Core

RTL interpreter

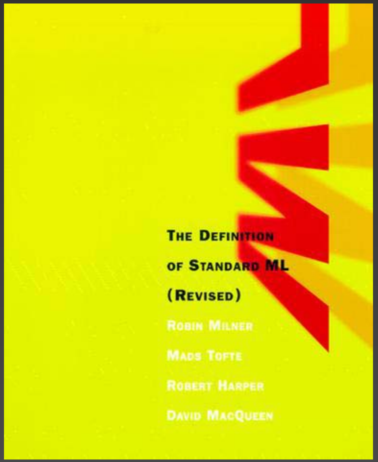
# x86 Abstract Syntax (~600 lines)

```
Inductive operand : Set := ...

Inductive instr : Set :=
| AAA : instr
| AAD : instr
| AAM : instr
| AAS : instr
| ADC : ∀ (width:bool) (op1 op2:operand), instr
| ADD : ∀ (width:bool) (op1 op2:operand), instr
| AND : ∀ (width:bool) (op1 op2:operand), instr
| ...
 (* 300 lines later *)
| XLAT : instr
| XOR : ∀ (op1 op2:operand), instr.
```

# *How to give semantics?*

- Usual approach is to use some form of structured operational semantics on the AST.
  - c.f., CESK machine, or TAL-like machine
  - encoded using an inductively defined predicate

- Worked in the 90's when we were doing languages on paper.



- But this doesn't scale…

# *Problems with SOS in Coq*

1. Rule explosion (e.g., for error cases).
2. No exhaustiveness checking.
3. Difficult to extend with new features (e.g., state).
4. Difficult to re-use results across models.
5. Coq won't symbolically reduce in proofs.
6. Can't (directly) extract executable code!

# *Instead:*

Define a core, orthogonal language

- typically, some monadic IL

Define translation from big language to this core.

- as a (total) function
- regain pattern matching, abstraction, re-use, etc.

Give small-step operational semantics for the core.

- ideally, define step as an executable function from machine states to (finite sets of) machine states.

# RTL Core Language (251 lines)

```
Module RTL(M : MACHINE_SIG).
  Include M.
  ...
  Inductive rtl : Set :=
  | arith_r : ∀s (b:bit_vector_op)(r1 r2:vreg s)(rd:vreg s), rtl
  | test_r : ∀s (top:test_op)(r1 r2:vreg s)(rd:vreg 1), rtl
  | if_r : vreg size1 -> rtl -> rtl
  | cast_s_r : ∀s1 s2 (r1:vreg s1) (rd:vreg s2), rtl
  | cast_u_r : ∀s1 s2 (r1:vreg s1) (rd:vreg s2), rtl
  | load_imm_r : ∀s (i:int s) (rd:vreg s),  rtl
  | set_loc_r : ∀s (rs:vreg s) (l:location s), rtl
  | get_loc_r : ∀s (l:location s) (rd:vreg s), rtl
  | set_byte_r : ∀ (rs:vreg 8)(addr:vreg size_addr), rtl
  | get_byte_r : ∀ (addr:vreg size_addr)(rd:vreg 8), rtl
  | choose_r : ∀s (rd:vreg s), rtl
  | error_r : rtl
  | safe_fail_r : rtl.
  ...
  Definition RTL_state := { rtl_mach:mach_state ; rtl_oracle : … }

  Definition interp (r:rtl) : State RTL_state unit := ...

End RTL.
```

# Non-Determinism through Oracle

```
Module RTL(M : MACHINE_SIG).
  Include M.
  ...
  Inductive rtl : Set :=
  | arith_r : ∀s (b:bit_vector_op)(r1 r2:vreg s)(rd:vreg s), rtl
  | test_r : ∀s (top:test_op)(r1 r2:vreg s)(rd:vreg 1), rtl
  | if_r : vreg size1 -> rtl -> rtl
  | cast_s_r : ∀s1 s2 (r1:vreg s1) (rd:vreg s2), rtl
  | cast_u_r : ∀s1 s2 (r1:vreg s1) (rd:vreg s2), rtl
  | load_imm_r : ∀s (i:int s) (rd:vreg s),  rtl
  | set_loc_r : ∀s (rs:vreg s) (l:location s), rtl
  | get_loc_r : ∀s (l:location s) (rd:vreg s), rtl
  | set_byte_r : ∀ (rs:vreg 8)(addr:vreg size_addr), rtl
  | get_byte_r : ∀ (addr:vreg size_addr)(rd:vreg 8), rtl
  | choose_r : ∀s (rd:vreg s), rtl
  | error_r : rtl
  | safe_fail_r : rtl.
  ...
  Definition RTL_state := { rtl_mach:mach_state ; rtl_oracle : … }

  Definition interp (r:rtl) : State RTL_state unit := ...

End RTL.
```

# Translation and Stepping (3,500 lines)

```
Definition instr_to_rtl (p:prefix)(ins:x86.instr) :=
  match ins with
  | AAA => conv_AAA p i
  | ADC w op1 op2 => conv_ADC p w op1 op2
  | ADD w op1 op2 => conv_ADC p w op1 op2
  | ...
  end.


Definition step : State RTL_state unit :=
  pc <- get_loc pc_loc ;
  [pre,instr,length] <- fetch_instruction pc ;
  let default_new_pc := pc + length in
  set_loc pc_loc default_new_pc ;;
  RTL_step_list (instr_to_rtl pre instr).
```

# Translation of ADD instruction

```
Definition conv_ADD prefix mode op1 op2 :=
let load := load op prefix mode in
let set := set op prefix mode in
let seg := get segment op2 prefix DS op1 op2 in
  zero ← load Z size1 0;
  up ← load Z size1 1;
  p0 ← load seg op1;
  p1 ← load seg op2;
  p2 ← arith add p0 p1;    (* real work here *)
  set seg p2 op1;;
  b0 ← test lt zero p0;
  b1 ← test lt zero p1;
  b2 ← test lt zero p2;
  b3 ← arith xor b0 b1;
  b3 ← arith xor up b3;
  b4 ← arith xor b0 b2;
  b4 ← arith and b3 b4;
  set flag OF b4;; ...
```

# *Execution Summary*

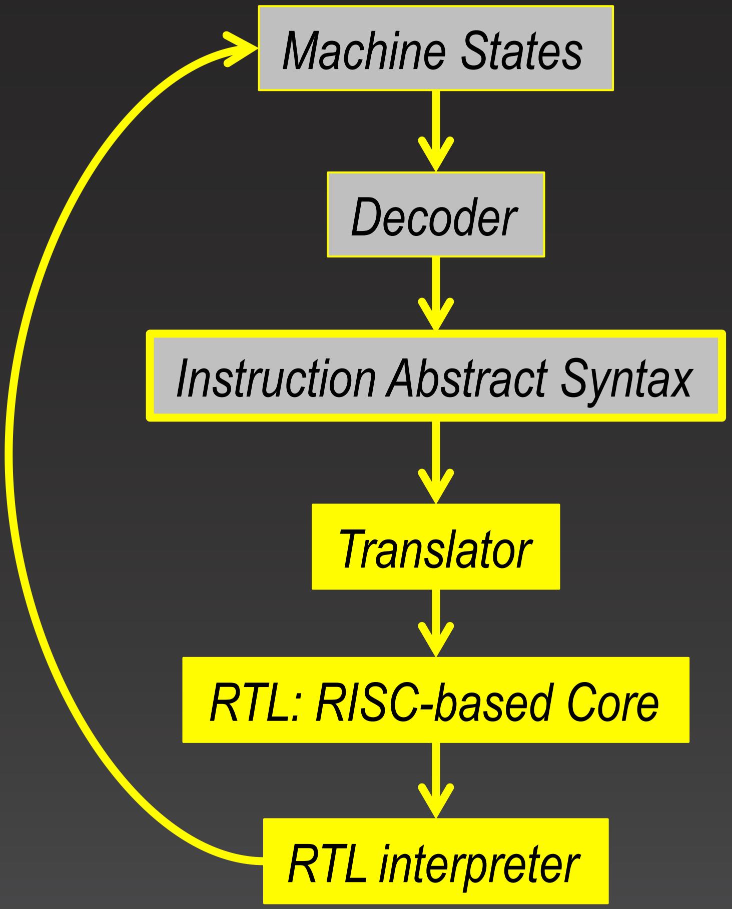
Generic RTL functor

- abstracts machine state
- simple, core RISC instructions
- functional interpreter, easy to extract executable code
- non-determinism modeled with oracle
- relatively easy to reason about

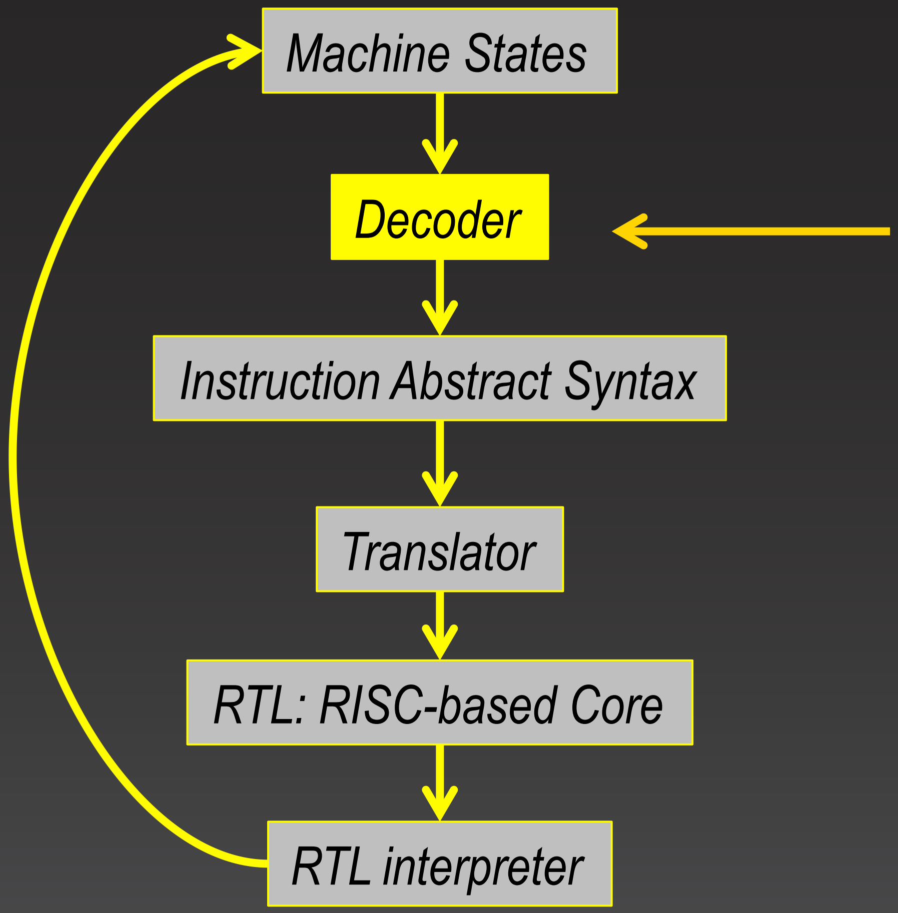
Translation into RTL

- can essentially follow the definitions in the manual
- type-classes, notation, and monads crucial
- but this is where most of the bugs lurk

# Our x86 Model in Coq

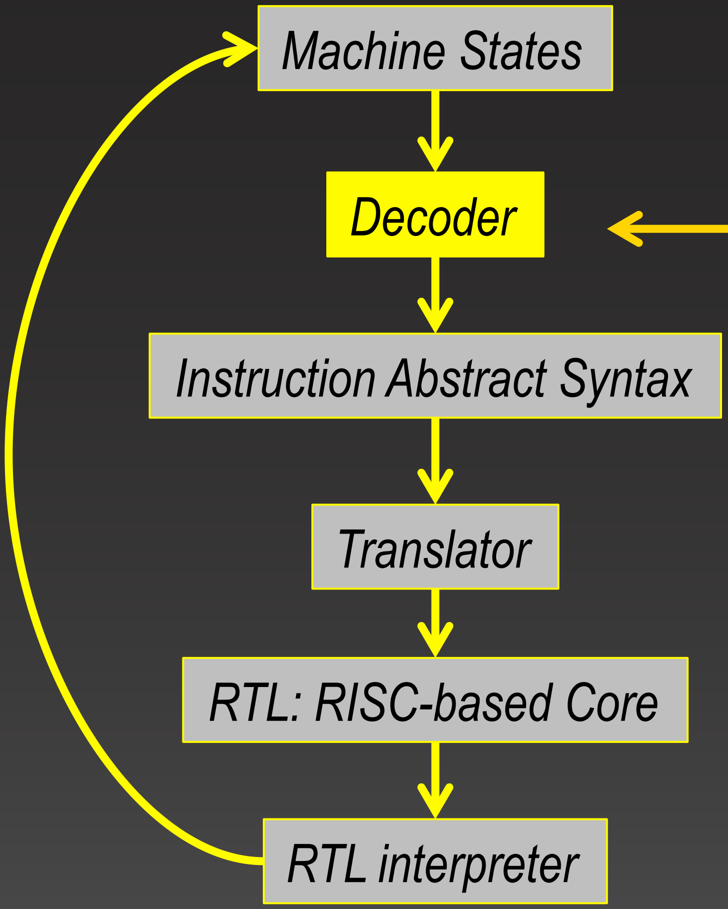


# Our x86 Model in Coq



Now for decoding....

# Our x86 Model in Coq



Machine States

Decoder

Instruction Abstract Syntax

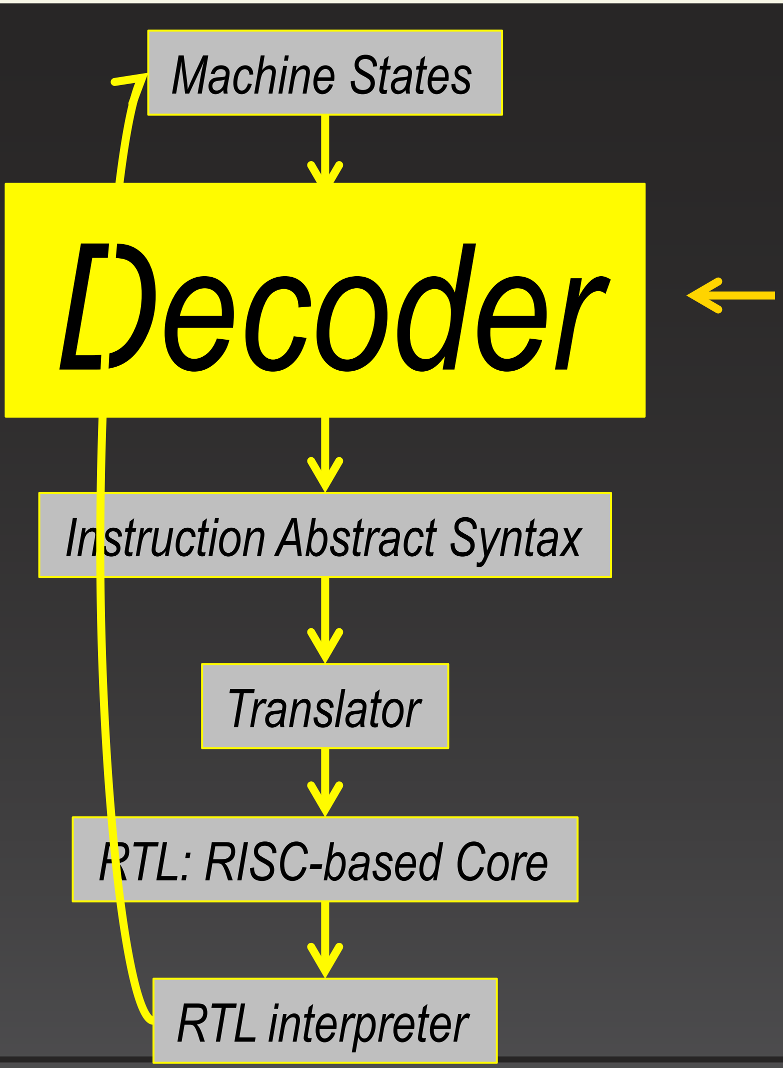
Translator

RTL: RISC-based Core

RTL interpreter

How hard can that be?

# Our x86 Model in Coq


Machine States

Decoder

Instruction Abstract Syntax

Translator

RTL: RISC-based Core

RTL interpreter

Harder than I thought!!

# *Decoding*

For RISC architectures, decoding isn't that hard.

- can write a reasonable parser by hand.

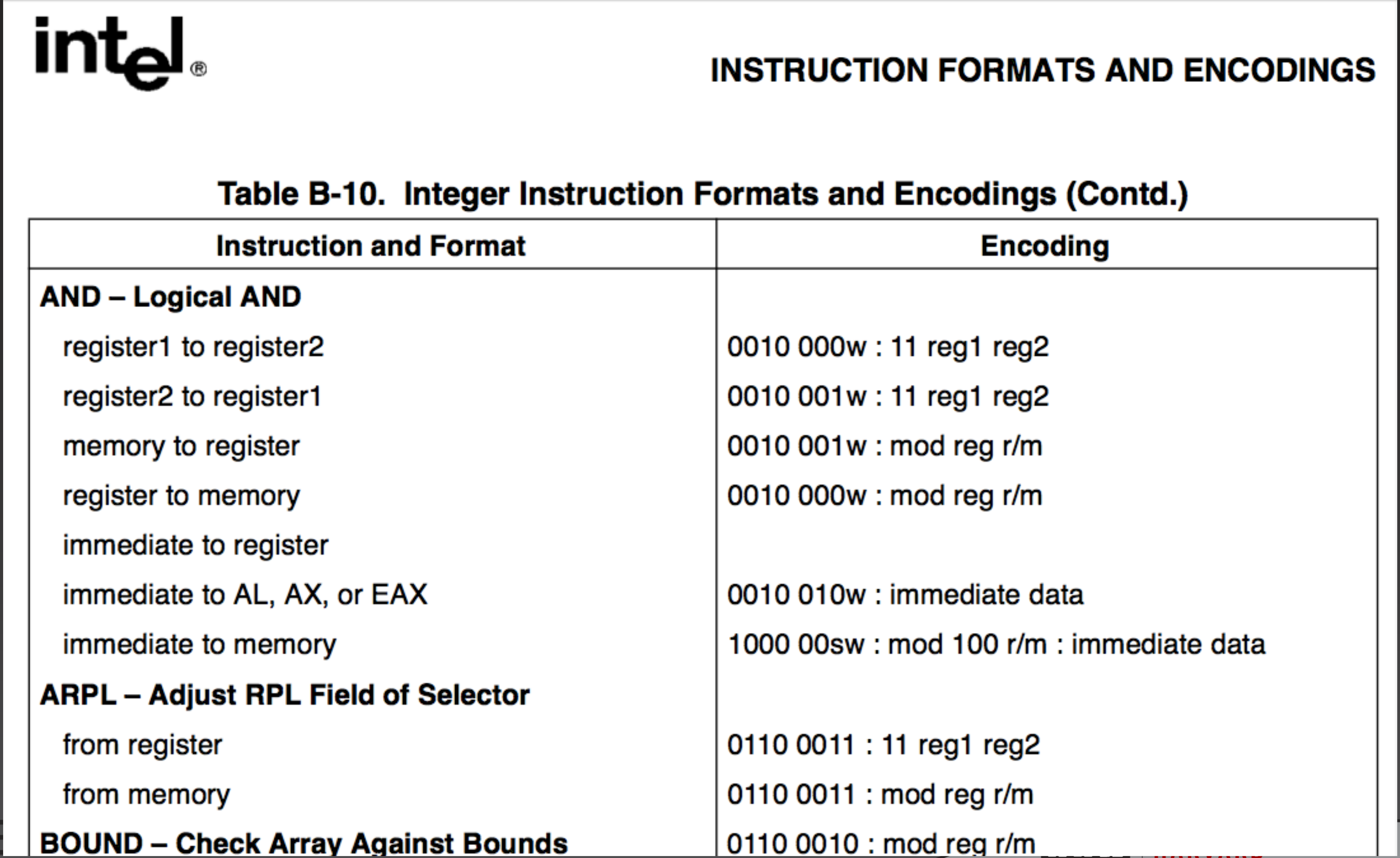
For x86, it's essentially impossible.

- thousands of opcodes, many addressing modes, etc.
- prefix bytes override things like size of constants
- the number of bytes depends upon earlier bytes seen and can range from 1 to 15.

Plus, we need to *reason* about parsing.

- need to relate regexps used in checker to model's decoder

# From the Intel Manual…

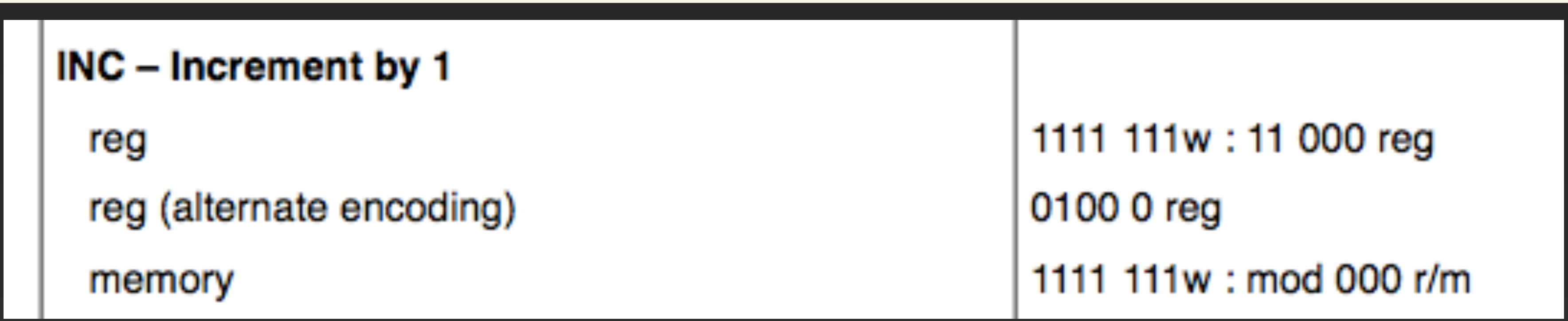


INSTRUCTION FORMATS AND ENCODINGS

**Table B-10. Integer Instruction Formats and Encodings (Contd.)**

| Instruction and Format | Encoding |
|---|---|
| **AND – Logical AND** | |
| register1 to register2 | 0010 000w : 11 reg1 reg2 |
| register2 to register1 | 0010 001w : 11 reg1 reg2 |
| memory to register | 0010 001w : mod reg r/m |
| register to memory | 0010 000w : mod reg r/m |
| immediate to register | |
| immediate to AL, AX, or EAX | 0010 010w : immediate data |
| immediate to memory | 1000 00sw : mod 100 r/m : immediate data |
| **ARPL – Adjust RPL Field of Selector** | |
| from register | 0110 0011 : 11 reg1 reg2 |
| from memory | 0110 0011 : mod reg r/m |
| **BOUND – Check Array Against Bounds** | 0110 0010 : mod reg r/m |

# Example Grammar for INC



| | |
|---|---|
| **INC – Increment by 1** | |
| reg | 1111 111w : 11 000 reg |
| reg (alternate encoding) | 0100 0 reg |
| memory | 1111 111w : mod 000 r/m |

```
Definition INC_g : grammar instr :=
    "1111" $$ "111" $$ bit $ "11000" $$ reg @
    (fun (w,r) => INC w (Reg_op r))
||  "0100" $$ "0" $$ reg @
    (fun r => INC true (Reg_op r)
||  "1111" $$ "111" $$ bit $ (emodrm "000") @
    (fun (w,op1) => INC w op1).
```

# Regular Grammar DSL

```
Inductive grammar : Type -> Type
| Char : char -> grammar char
| Eps : grammar unit
| Cat : ∀T U, grammar T -> grammar U -> grammar (T*U)
| Zero : ∀T, grammar T
| Alt : ∀T U, grammar T -> grammar U -> grammar (T+U)
| Star : ∀T, grammar t -> grammar (list T)
| Map : ∀T U, grammar T -> (T -> U) -> grammar U

Infix “+” := Alt.
Infix “$” := Cat.
Infix “@” := Map.
Infix “||” := (fun g1 g2 => g1 + g2 @
 (fun v => match v with inl x => x | inr y => y end))
Infix “$$” := (fun x y => x $ y @ snd).
```

# Grammar Semantics

[[ ]] : grammar T -> (string * T) -> Prop.

[[Eps]] = {(nil, tt)}
[[Zero]] = {}
[[Char c]] = {(c::nil, c)}
[[$g_1$+$g_2$]] = {(s,inl v) | (s,v) in [[$g_1$]]}
U {(s,inr v) | (s,v) in [[$g_2$]]}
[[$g_1$\$$g_2$]] =
{($s_1$++$s_2$,($v_1$,$v_2$)) | ($s_i$,$v_i$) in [[$g_i$]]}
[[g*]] = [[Eps]] U
{(s,v) | s≠nil /\ s in [[g\$g*]]}
[[g@f]] = {(s, f v) | (s,v) in [[g]]}

# *Typed Grammars as Specs*

The grammar language is very attractive for specification:

- typed “semantic actions”
- easy to build new combinators
- easy transliteration from the Intel manual

Unlike Yacc/Flex/etc., has a good semantics:

- easy inversion principles
- good algebraic properties
- e.g., easy to refactor or optimize grammar

# *Executable Decoding*

But alas, the semantics as given isn’t executable.

Approaches we tried:

- Haskell-style parsing combinators (bad)
- PEG (not compositional)
- Online derivative-based parser (okay)
- Table-driven parser based on careful phase-split of the online derivative approach (in progress).

# Derivative-Based Parsing

```
deriv c g = {(s,v) | (c::s,v) in [[g]]}

extract g = {v | (nil,v) in [[g]]}

parse g (c::s) := parse (deriv c g) s
parse g nil := extract g


Theorem:
  In v (parse g cs) <-> (cs,v) in [[g]].
```

# Derivatives for Regular Expressions

```
(* deriv c g = {s | c::s in [g]} *)

deriv c (Char c) = Eps
deriv c (g1 + g2) =  deriv c g1 + deriv c g2
deriv c (g*) = (deriv c g $ g*)
deriv c (g1 $ g2) =
   (deriv c g1 $ g2) + (null g1 $ deriv c g2)
deriv c _ = Zero
```

# Derivatives for Grammars

```
(* deriv c g = {(s,v) | (c::s,v) in [g]} *)

deriv c (Char c) = Eps @ (fun _ => c)
deriv c (g1 + g2) =  deriv c g1 + deriv c g2
deriv c (g*) = (deriv c g $ g*) @ (::)
deriv c (g1 $ g2) =
   (deriv c g1 $ g2) || (null g1 $ deriv c g2)
deriv c (g @ f) = (deriv c g) @ f
deriv c _ = Zero
```

# *Notes on Derivatives*

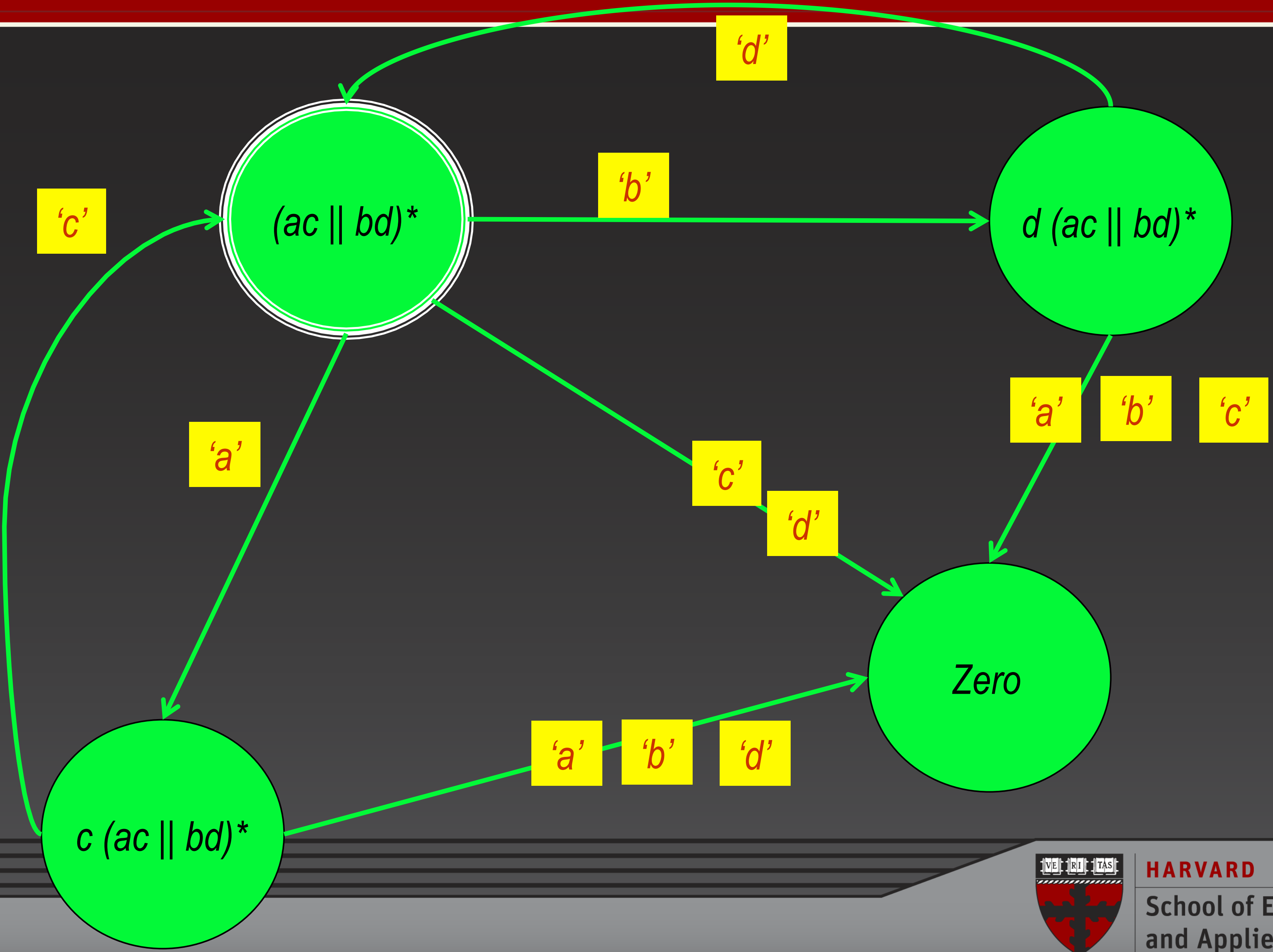
- Old idea due to Brzozowski (1964), revitalized by Reppy et al., and extended by Might.
- Avoids reasoning about automata (graphs).
- In practice, we must optimize the grammars as we construct them:

```
Eps $ g → g @ (fun x => (tt,x))
Zero $ g → Zero
Zero || g → g @ inr
g @ f1 @ f2 → g @ (f1 o f2)
...
```

# *Table-Based Recognition*

- The parser I showed you is calculating derivatives on-line.
- Brzozowski showed how to construct a DFA from a regular *expression* using derivatives.
  - calculate (`deriv c r`) for each `c` in the alphabet.
  - each unique (up to the optimizations) derivative corresponds to a state.
  - continue by calculating all reachable states' derivatives.
  - guaranteed this process will terminate!

# Example: (ac || bd)*



HARVARD
School of Engineering
and Applied Sciences

# *Bad News*

The derivatives for regular *expressions* are finite.

But as defined, we can have an unbounded number of derivatives for our typed, regular *grammars*.

This seems to preclude a table-based parser where we calculate all of the derivatives up front.

# *Breaking Finite Derivatives*

For regular expressions:

```
deriv a (a*) =
  (deriv a a) $ a* = Eps $ a* = a*
```

# *Breaking Finite Derivatives*

For regular expressions:

```
deriv a (a*) =
  (deriv a a) $ a* = Eps $ a* = a*
```

For regular grammars:

```
deriv a (a*) =
   (deriv a a) $ a* @ (::) =
   (Eps @ (λ_ => a)) $ a* @ (::) =
  a* @ (λ x => a::x)
```

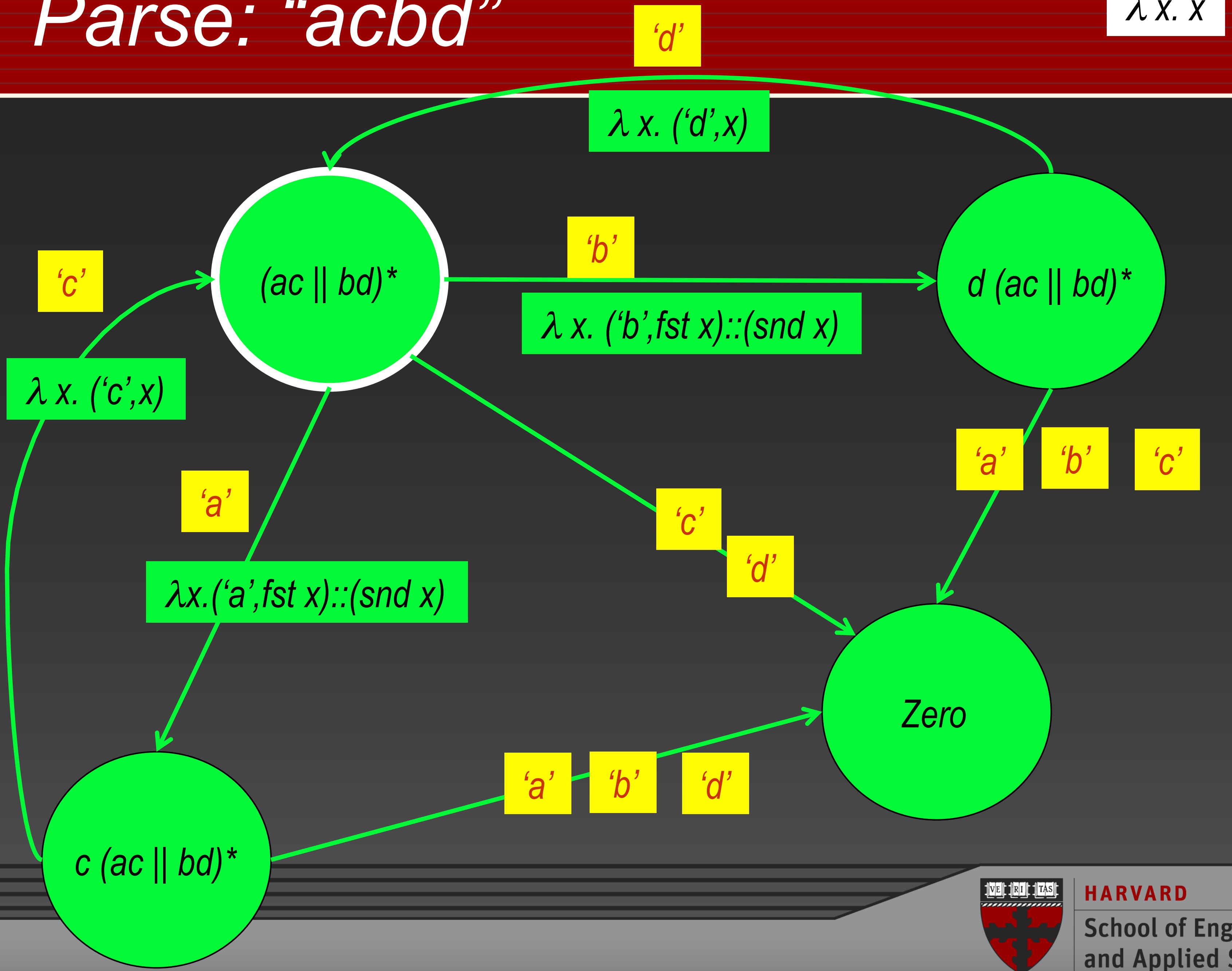
# *Regaining Finite Derivatives*

The solution is to split grammars into a map-free grammar and a single mapping function.

```
split: grammar T ->
        {a : ast_gram & (ast_tipe a) -> T}
```

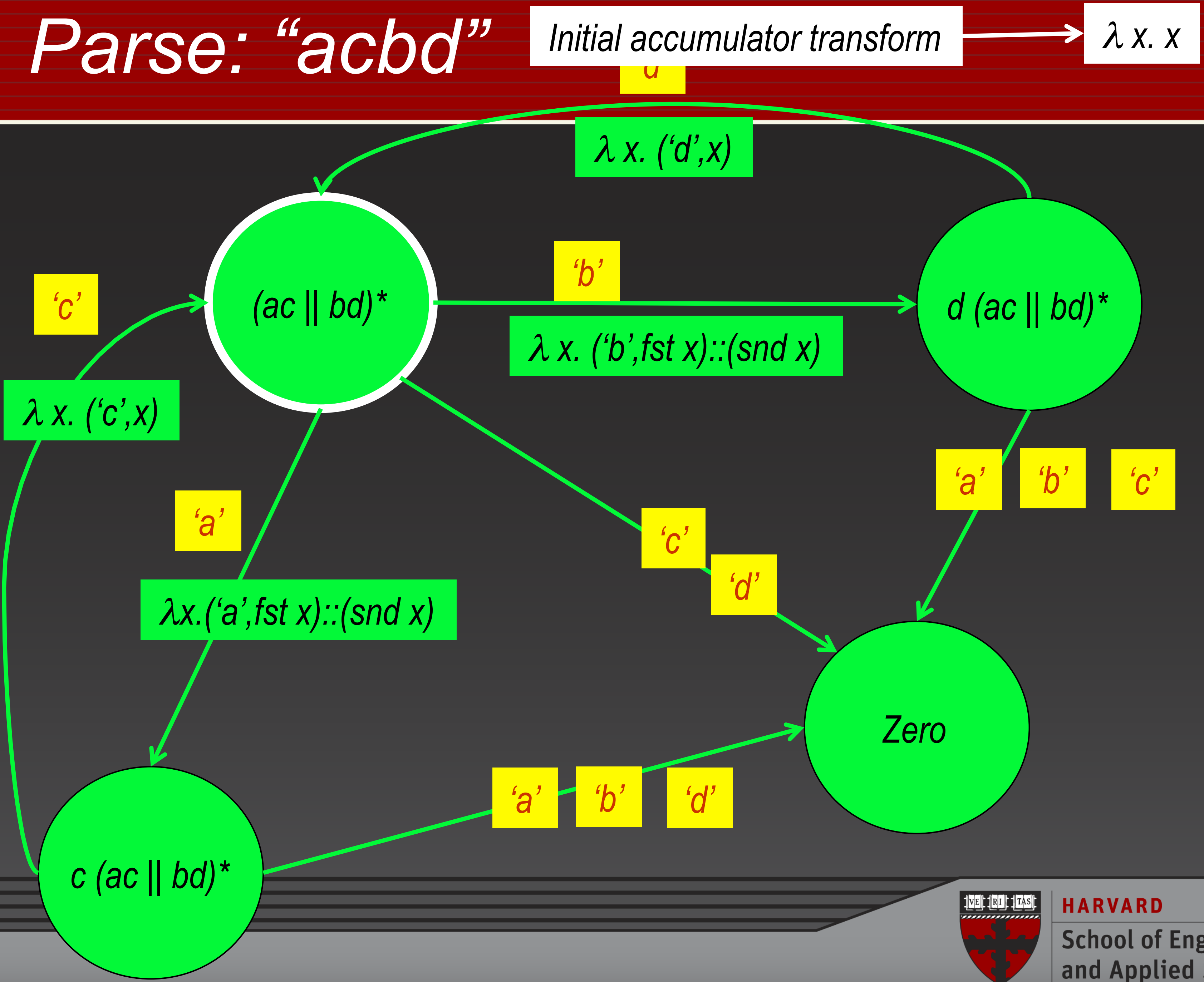
- As we calculate derivatives, we continue to split.
  - the states correspond to AST grammars
  - the *edges* are labeled with the maps
  - the parser computes a composition of maps

# Parse: “acbd”

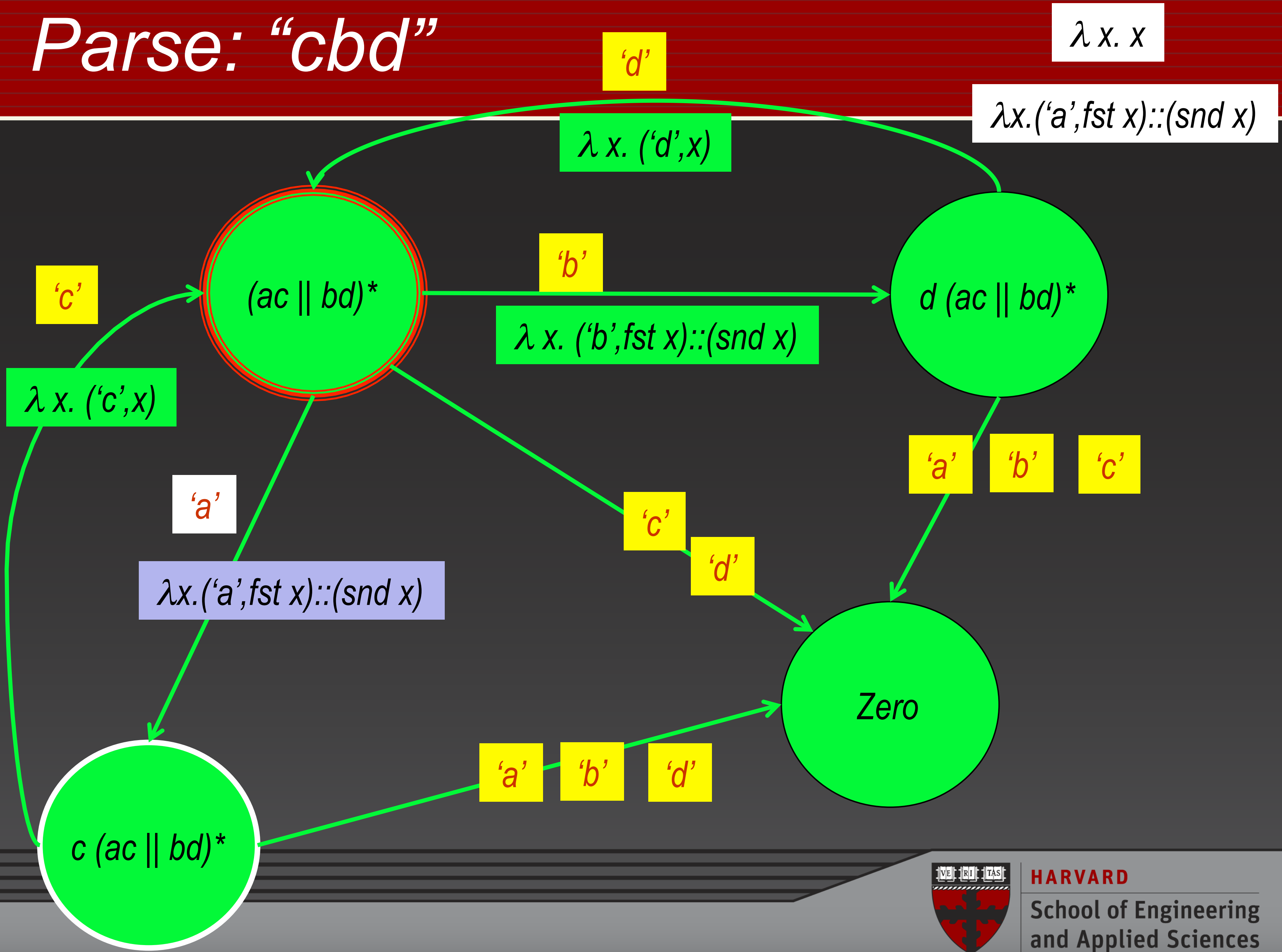

λ x. x
‘d’
λ x. (‘d’,x)
‘c’
(ac || bd)*
‘b’
d (ac || bd)*
λ x. (‘b’,fst x)::(snd x)
λ x. (‘c’,x)
‘a’
‘b’
‘c’
‘a’
‘c’
‘d’
λx.(‘a’,fst x)::(snd x)
Zero
‘a’
‘b’
‘d’
c (ac || bd)*

# Parse: “acbd”

Initial accumulator transform → λ x. x

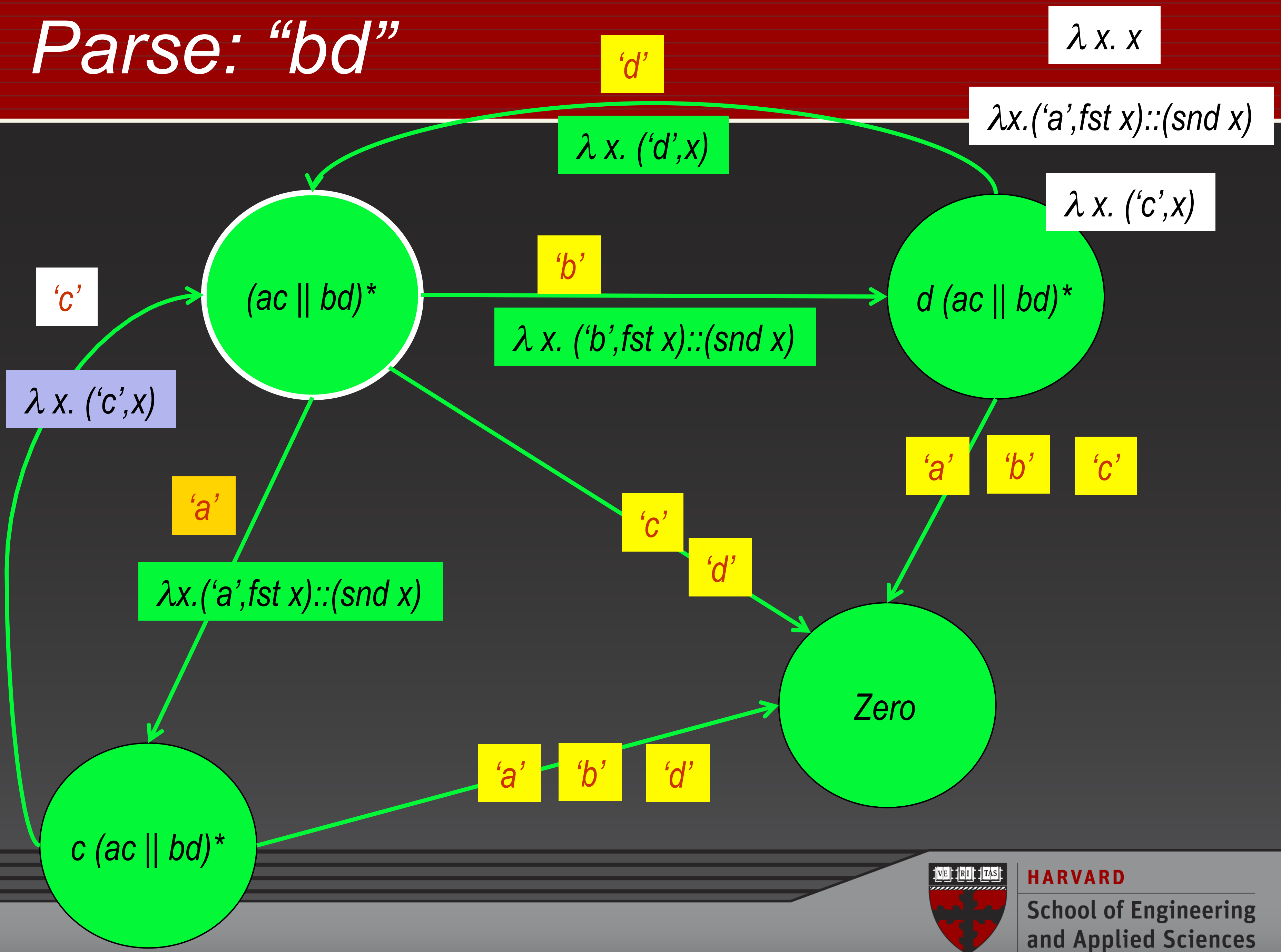


# Parse: “cbd”



HARVARD
School of Engineering
and Applied Sciences

# Parse: “bd”



λ x. x

λx.(‘a’,fst x)::(snd x)

λ x. (‘c’,x)

‘d’

λ x. (‘d’,x)

(ac || bd)*

‘b’

λ x. (‘b’,fst x)::(snd x)

d (ac || bd)*

‘c’

λ x. (‘c’,x)

‘a’

λx.(‘a’,fst x)::(snd x)

‘c’

‘d’

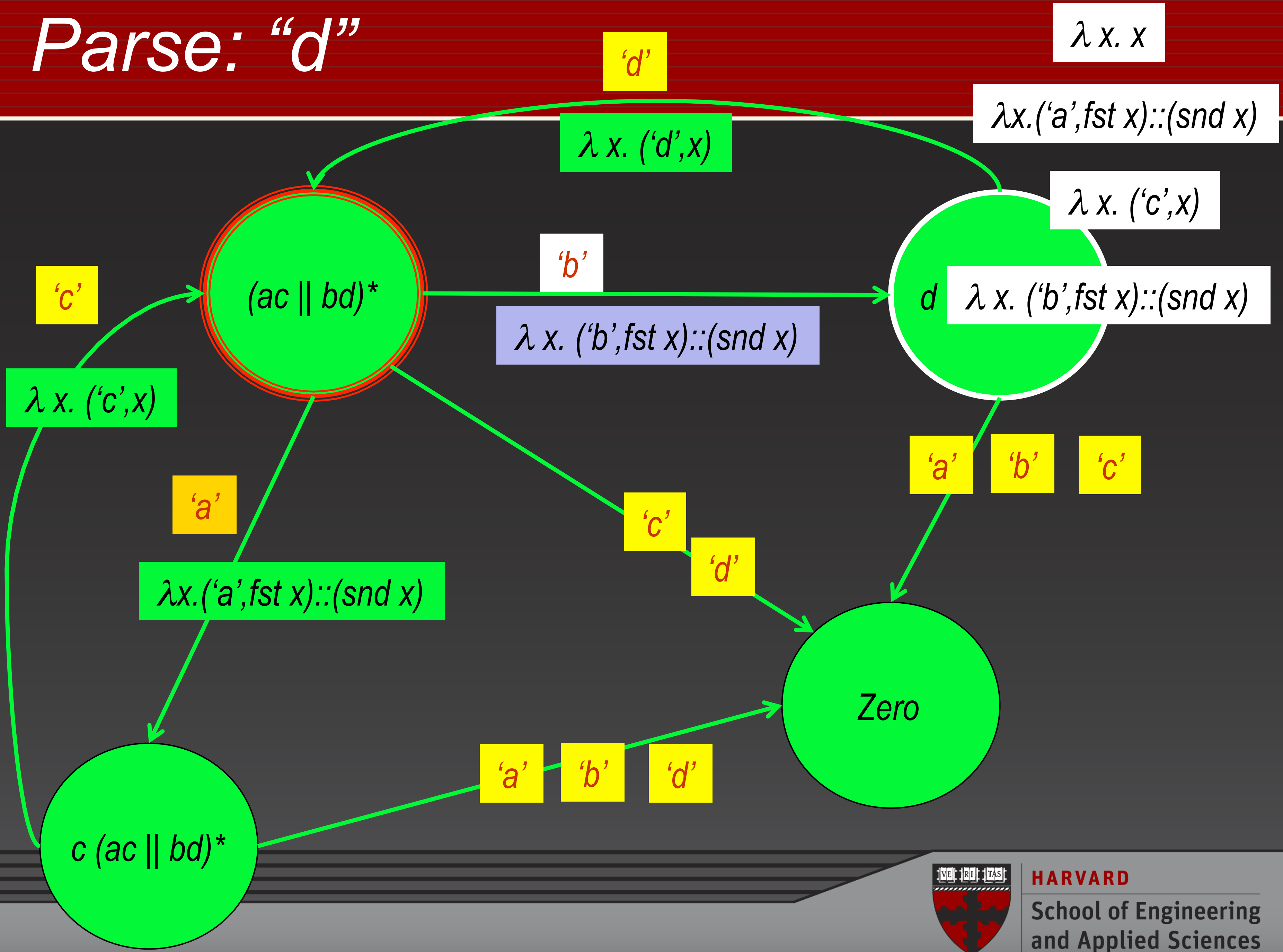
‘a’ ‘b’ ‘c’

Zero

‘a’ ‘b’ ‘d’

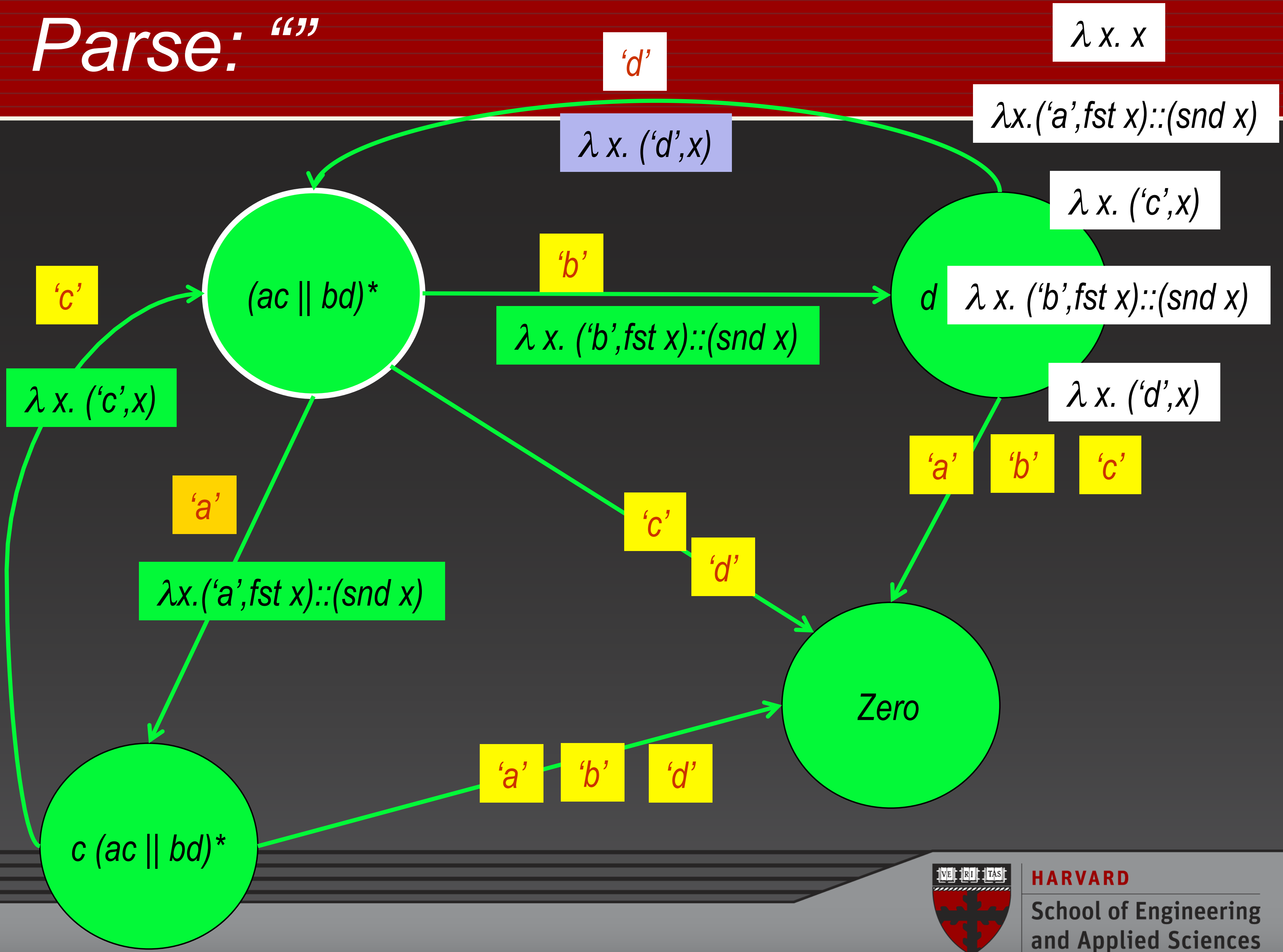
c (ac || bd)*

# Parse: “d”

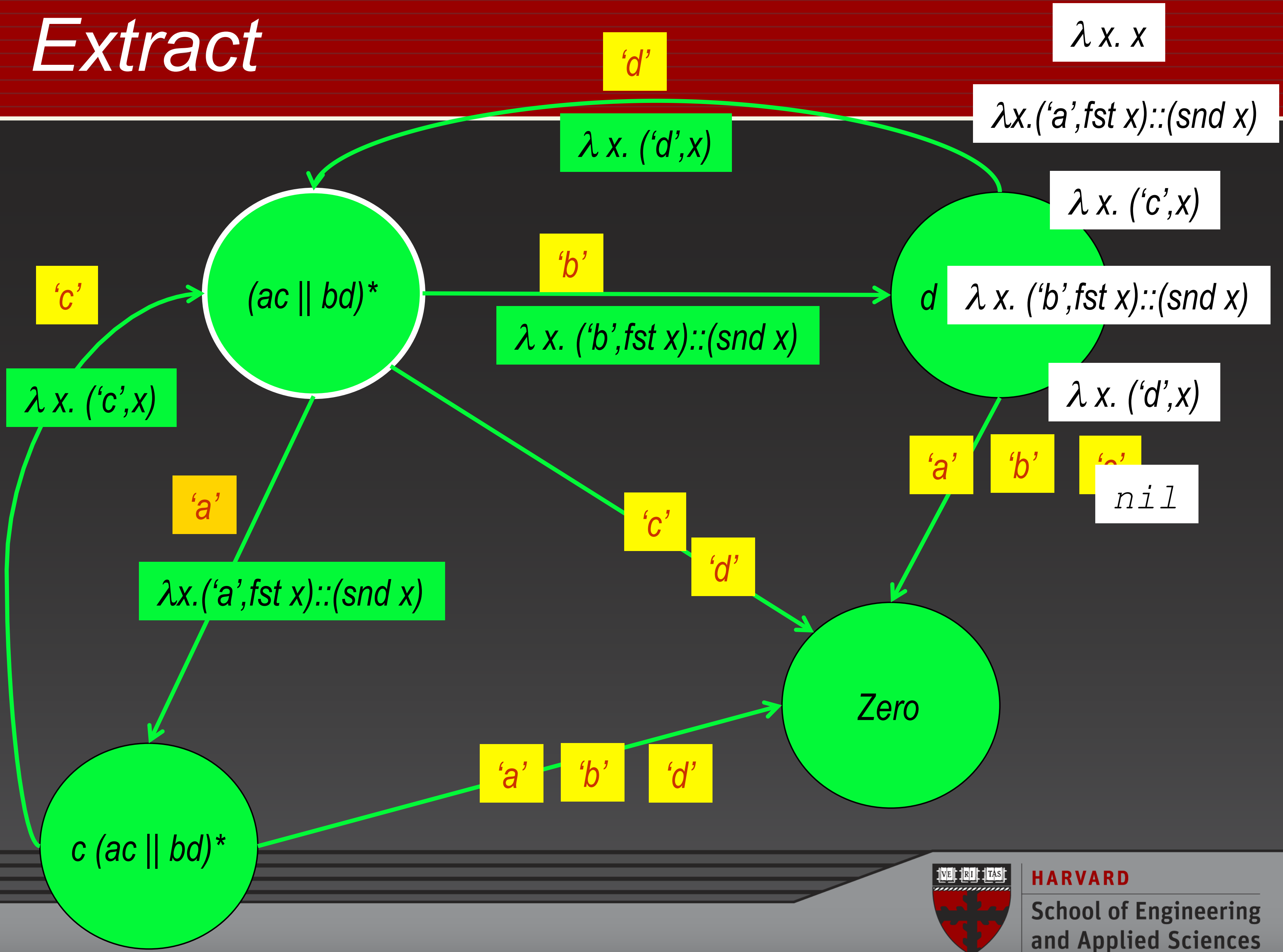


HARVARD
School of Engineering
and Applied Sciences

# Parse: “”


λ x. x
‘d’
λx.(‘a’,fst x)::(snd x)
λ x. (‘d’,x)
λ x. (‘c’,x)
‘b’
‘c’
(ac || bd)*
d
λ x. (‘b’,fst x)::(snd x)
λ x. (‘b’,fst x)::(snd x)
λ x. (‘c’,x)
λ x. (‘d’,x)
‘a’
‘b’
‘c’
‘a’
‘c’
‘d’
λx.(‘a’,fst x)::(snd x)
Zero
‘a’
‘b’
‘d’
c (ac || bd)*

# Extract



λ x. x

λx.('a',fst x)::(snd x)

λ x. ('c',x)

λ x. ('b',fst x)::(snd x)

λ x. ('d',x)

nil

'd'

λ x. ('d',x)

'c'

λ x. ('c',x)

(ac || bd)*

'b'

λ x. ('b',fst x)::(snd x)

d

'a'

λx.('a',fst x)::(snd x)

'c'

'd'

'a'

'b'

'c'

Zero

c (ac || bd)*

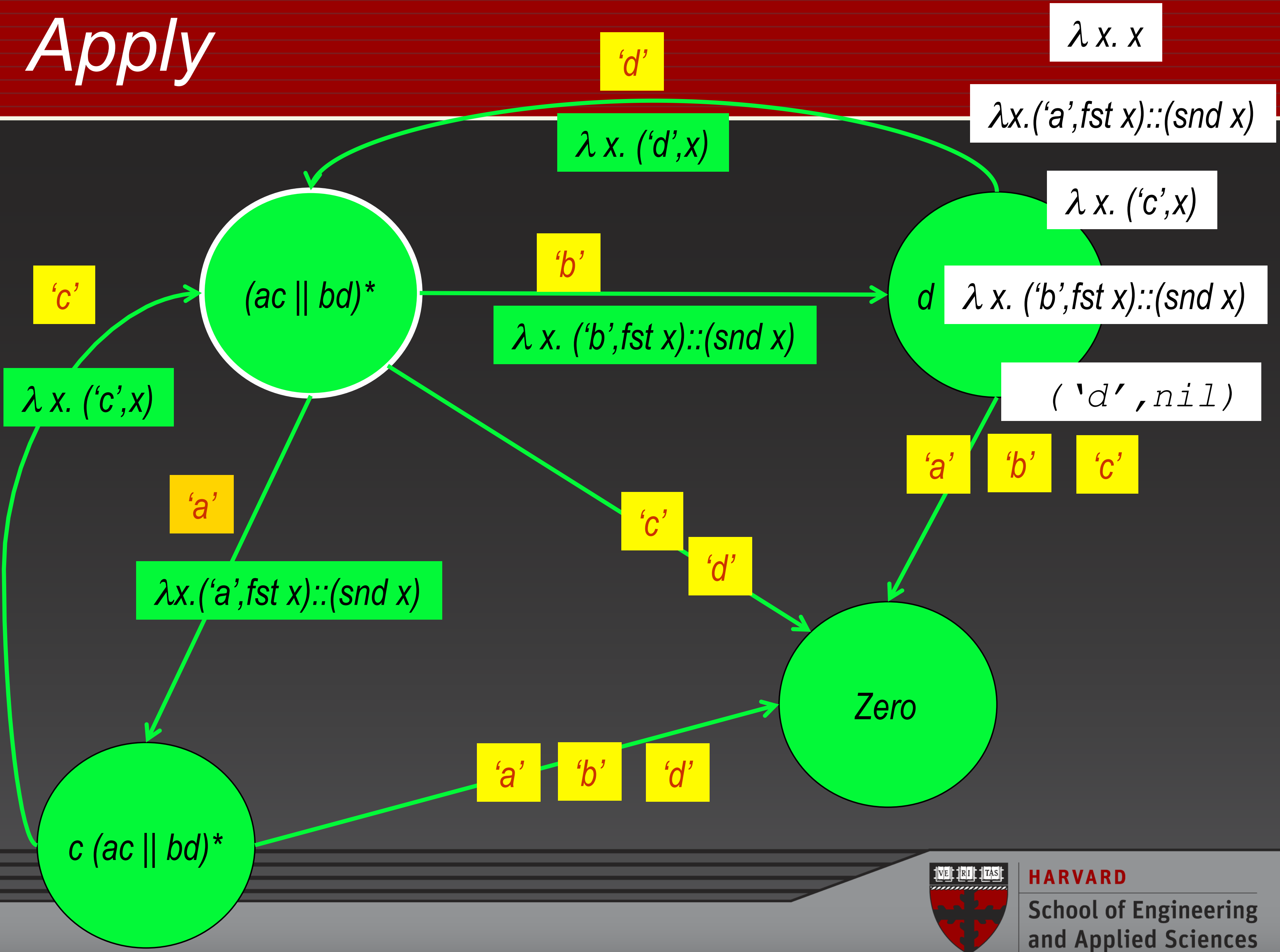
'a'

'b'

'd'

HARVARD
School of Engineering
and Applied Sciences

# Apply



(ac || bd)*

d

Zero

c (ac || bd)*

‘d’

λ x. (‘d’,x)

‘b’

λ x. (‘b’,fst x)::(snd x)

‘c’

λ x. (‘c’,x)

‘a’

λx.(‘a’,fst x)::(snd x)

‘c’ ‘d’

‘a’ ‘b’ ‘d’

‘a’ ‘b’ ‘c’

λ x. x

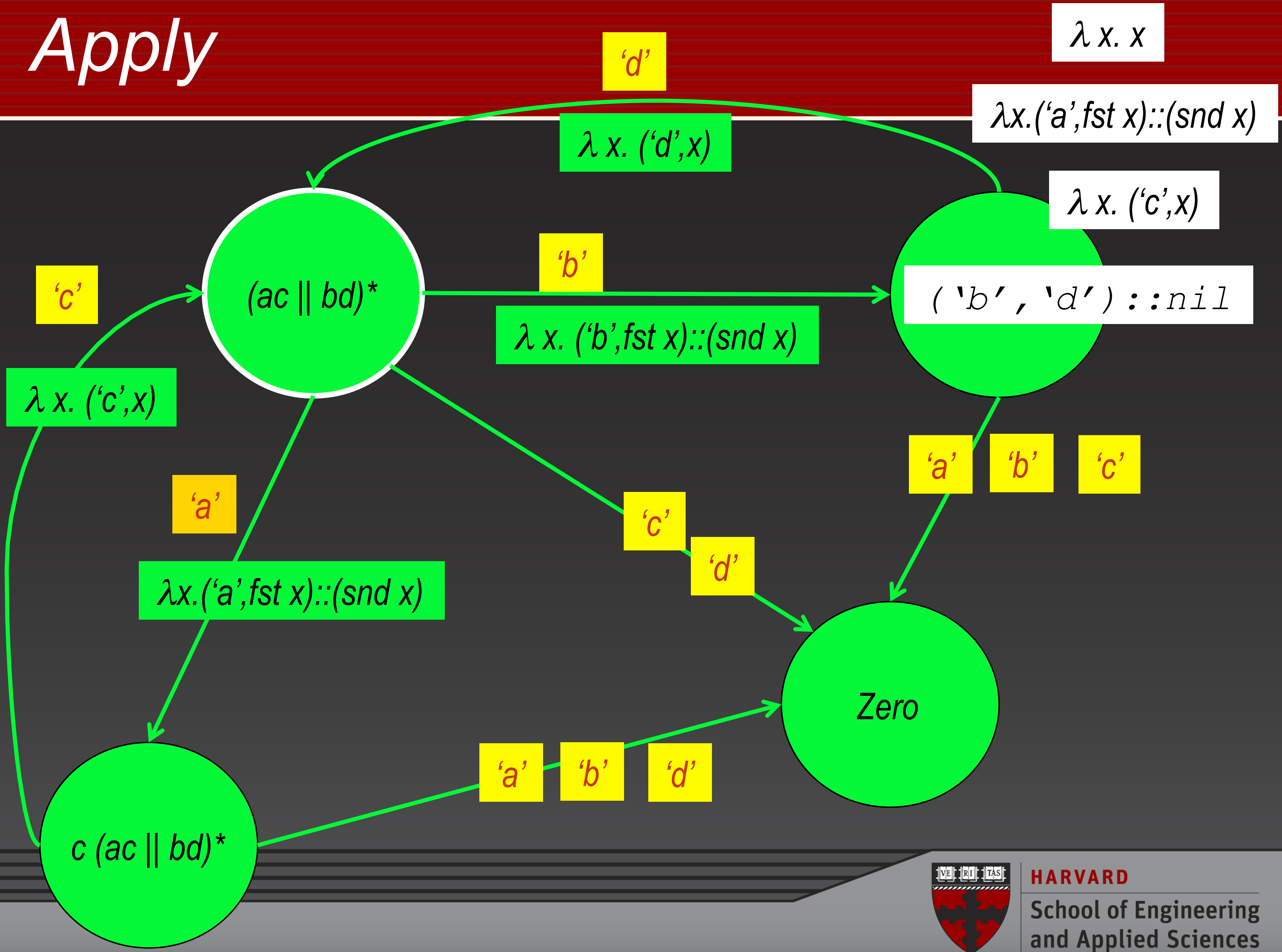
λx.(‘a’,fst x)::(snd x)

λ x. (‘c’,x)

λ x. (‘b’,fst x)::(snd x)

`('d',nil)`

# Apply



λ x. x

λx.(‘a’,fst x)::(snd x)

λ x. (‘c’,x)

`(‘b’,‘d’)::nil`

‘d’

λ x. (‘d’,x)

‘b’

λ x. (‘b’,fst x)::(snd x)

‘c’

λ x. (‘c’,x)

(ac || bd)*

‘a’

λx.(‘a’,fst x)::(snd x)

‘c’

‘d’

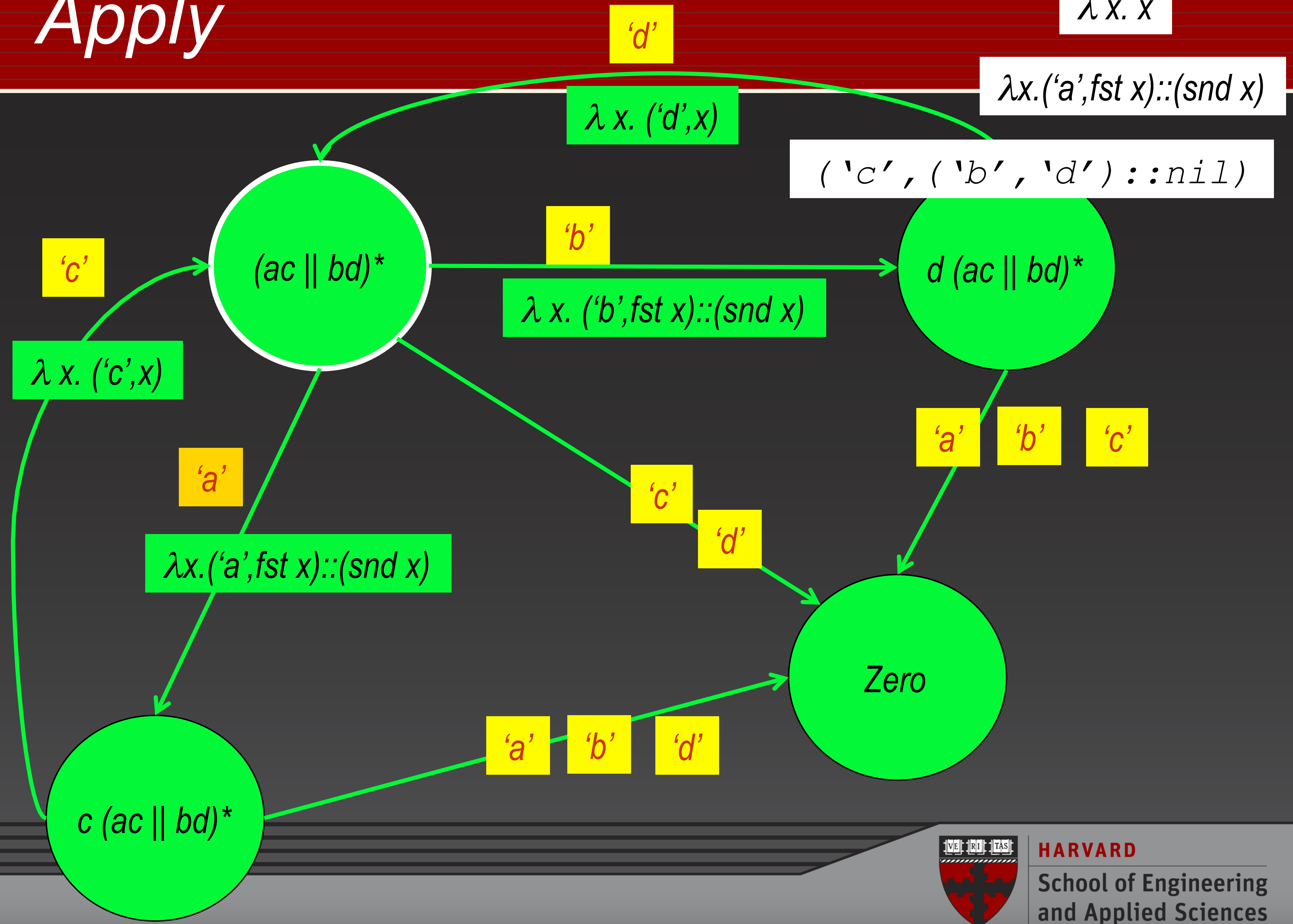
‘a’ ‘b’ ‘c’

Zero

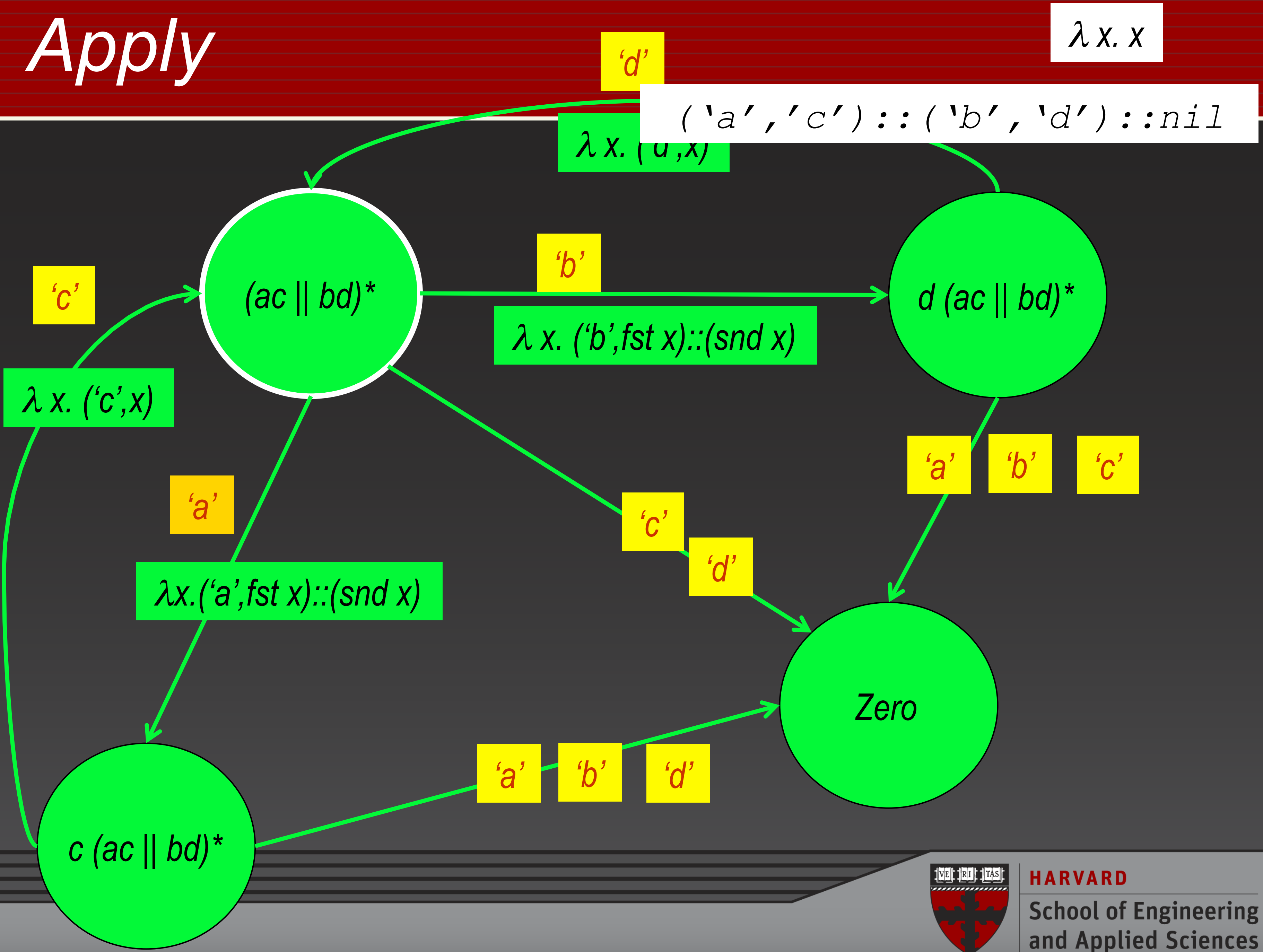
‘a’ ‘b’ ‘d’

c (ac || bd)*

# Apply

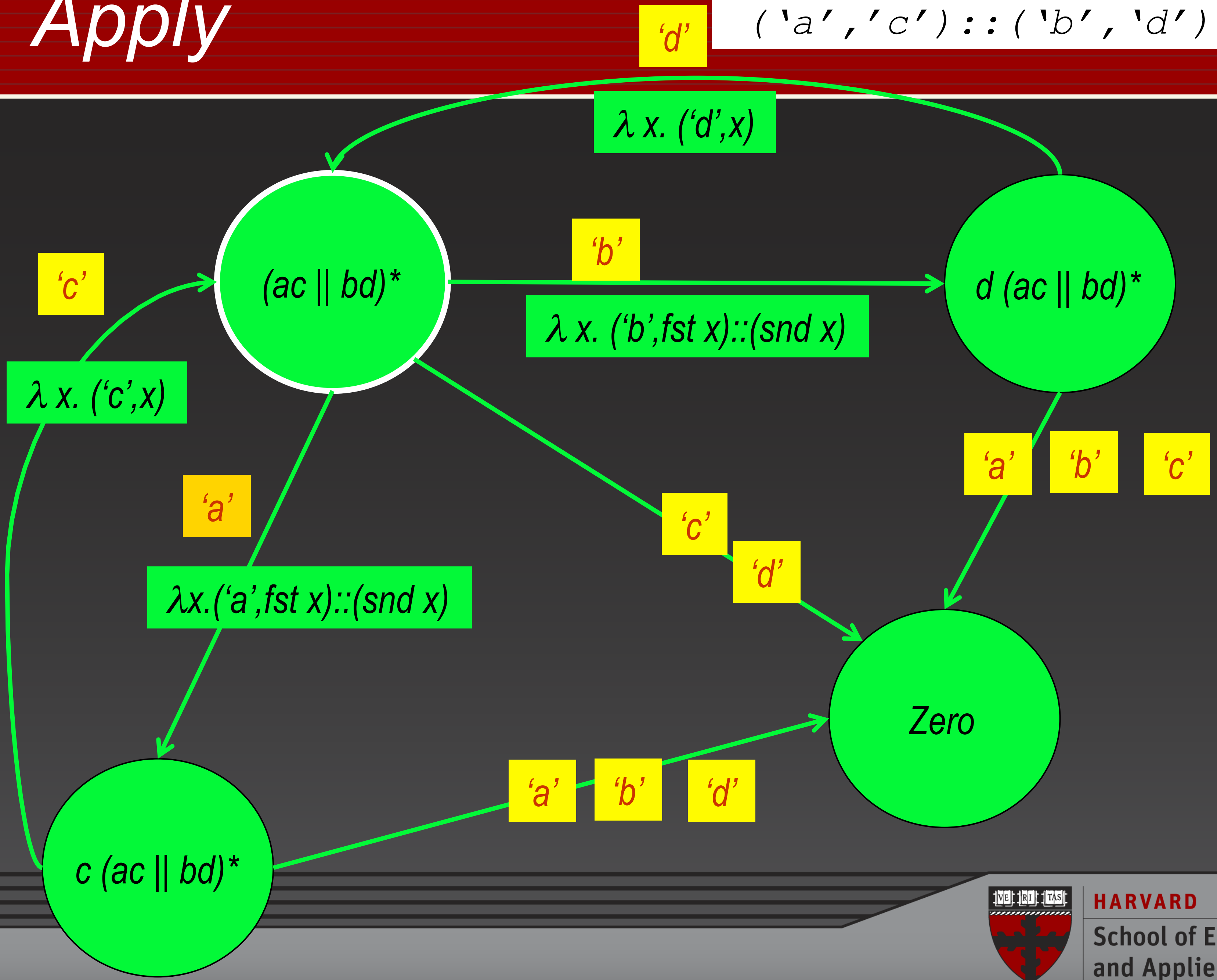

λ x. x
λx.('a',fst x)::(snd x)
'd'
λ x. ('d',x)
('c',('b','d')::nil)
'b'
'c'
(ac || bd)*
d (ac || bd)*
λ x. ('b',fst x)::(snd x)
λ x. ('c',x)
'a'
'b'
'c'
'a'
'c'
'd'
λx.('a',fst x)::(snd x)
Zero
'a'
'b'
'd'
c (ac || bd)*

# Apply


λ x. x
‘d’
('a','c')::('b','d')::nil
λ x. ( d ,x)
‘c’
(ac || bd)*
‘b’
d (ac || bd)*
λ x. (‘b’,fst x)::(snd x)
λ x. (‘c’,x)
‘a’
‘b’
‘c’
‘a’
‘c’
‘d’
λx.(‘a’,fst x)::(snd x)
Zero
‘a’
‘b’
‘d’
c (ac || bd)*

# Apply

`('a','c')::('b','d')::nil`



HARVARD
School of Engineering
and Applied Sciences

# *That’s nice!*

We can construct a table-driven parser by just calculating derivatives, and then splitting.

And it’s relatively easy to show that the parser is correct.

We can also use the table to determine if the grammar is ambiguous.

- any terminal state (i.e., that accepts the empty string) shouldn’t have alternatives.

# *Still Had Two Major Problems*

1. The semantic actions were too expensive.
   - for our table, each state corresponds to the 8th derivative
   - so each edge has the composition of 8 maps
   - solution: reflect internal transforms as a typed, sequent calculus and perform cut elimination.
   - Now the parser runs like a bat out of hell (100x faster than online derivatives.)
2. It literally took *days* to build the tables.
   - and this problem is fundamental to Coq…

# The Essence of the Problem

To optimize, we needed to represent terms as syntax.

```
Inductive tipe : Set :=
| Char : tipe
| Pair : tipe -> tipe -> tipe
| Sum : tipe -> tipe -> tipe
| ...

Inductive term : tipe -> tipe -> Set :=
| Id : ∀t, term t t
| Comp : ∀ t u v, term t u -> term u v -> term t v
| Fst : ∀t u, term (Pair t u) t
| ...
```

# The Essence of the Problem

To compile, we needed the syntax to be indexed by “tipes”.

```
Inductive tipe : Set :=
| Char : tipe
| Pair : tipe -> tipe -> tipe
| Sum : tipe -> tipe -> tipe
| ...

Inductive term : tipe -> tipe -> Set :=
| Id : ∀t, term t t
| Comp : ∀ t u v, term t u -> term u v -> term t v
| Fst : ∀t u, term (Pair t u) t
| ...
```

# The Compiler

```
Fixpoint interp (t:tipe) : Set :=
  match t with
  | Char => char
  | Pair t1 t2 => (interp t1) * (interp t2)
  | ...
  end

Fixpoint compile t u (e:term t u) : interp t -> interp u :=
  match e in term t u return interp t -> interp u with
  | Id t => (fun (x:interp t) => x)
  | Comp t u v f g =>
    let f_c := compile t u f in
    let g_c := compile u v g in
      fun (x:interp t) => g_c (f_c x)
  | ...
  end
```

# The Optimizer

```
Fixpoint opt_comp t u v (e1:term t u):
  term u v -> term t v  :=
    match e1 in term t u return term u v -> term t v with
    | Id t => (fun e2 => e2)
    | ...
    end

Fixpoint opt t u (e:term t u) : term t u :=
  match e in term t u return term t u with
  | Comp t u v f g =>
       opt_comp t u v (opt t u f) (opt u v g)
  | ...
  end

Lemma opt_corr : ∀t u (e:term t u),
    compile t u e = compile t u (opt t u e)
```

# *When you extract Ocaml code*

```
let rec opt_comp t u v (e1:term): term -> term :=
    match e1 with
     | Id t -> (fun e2 -> e2)
     | ...


let rec opt t u (e:term) : term :=
  match e with
   | Comp t u v f g ->
        opt_comp t u v (opt t u f) (opt u v g)
   | ...
```

OCaml can’t express the dependency of e’s type on the terms t and u.

# When you extract Ocaml code

```
let rec opt_comp t u v (e1:term): term -> term :=
    match e1 with
     | Id t -> (fun e2 -> e2)
     | ...


let rec opt t u (e:term) : term :=
  match e with
   | Comp t u v f g ->
       opt_comp t u v (opt t u f) (opt u v g)
   | ...
```

But extraction is too stupid to realize the tipes are not needed and could be erased.

# *When you extract Ocaml code*

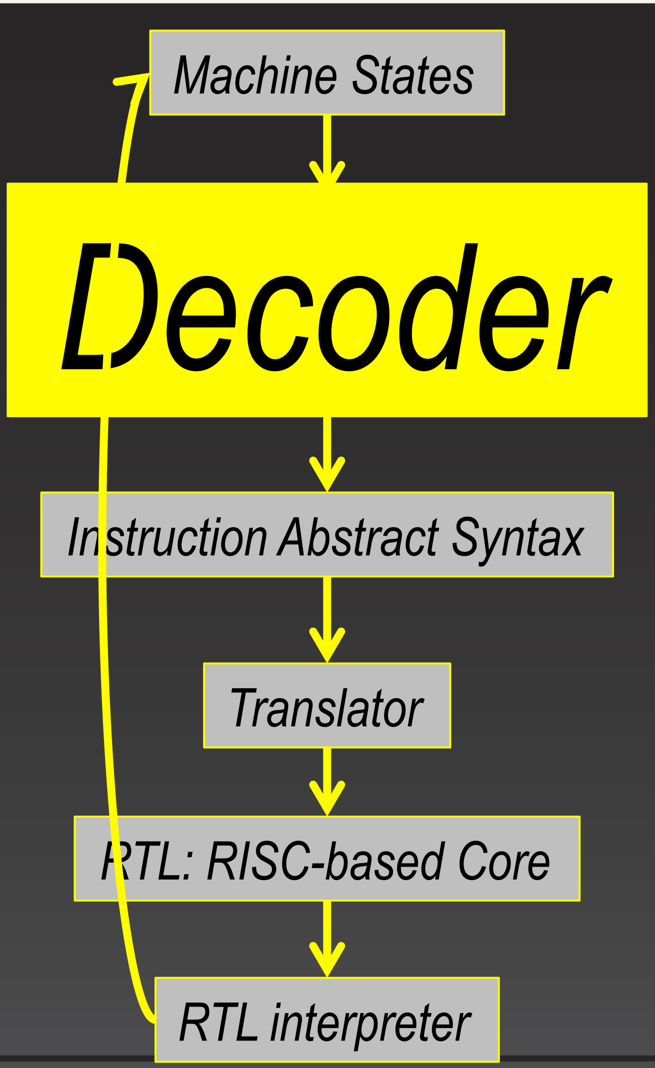
```
let rec opt_comp (e1:term): term -> term :=
    match e1 with
     | Id -> (fun e2 -> e2)
     | ...


let rec opt (e:term) : term :=
  match e with
   | Comp f g ->
        opt_comp (opt f) (opt g)
   | ...
```

There is an experimental feature (hack) in Coq that lets you get rid of unnecessary indices in the extracted code (Extraction Implicit.)

# *On Computational Irrelevance*

- Coq will automatically erase *its* types and proofs in the extracted code, but not terms like my “tipes”.
- We can’t use Coq’s types or proofs as because we lose injectivity for the type constructors and/or the ability to compile back to a Coq function.
- There are type theories (e.g., ICC*) that support a more principled approach to irrelevance.

# Our x86 Model in Coq



Machine States

Decoder

Instruction Abstract Syntax

Translator

RTL: RISC-based Core

RTL interpreter

← Whew!!!

# *Decoding Summary*

A nice declarative specification of the grammar.

- users can use arbitrary functions for semantic actions.
- can build nice notation/combinators.
- easy algebraic reasoning.

We can extract a provably-correct, table-driven parser that can be used for testing.

- but we have to use a hack.
- buried within here is compiler, optimizer, and a lot of proofs (~ 5Kloc)

# *Future Directions for x86 Model*

- Better validation
  - the parsing technology is aimed at building a faster model so we can do more testing/validation
- Extending the execution model
  - concurrency, system state, other architectures, …
- Extending the security policy
  - CFI, XFI, TAL, …
- Beyond regular grammars
  - e.g., CFGs

# Perhaps?

Break the DSLs out as first-class citizens.

- Provide mappings into ACL2, HOL, Coq, etc.
- Would be nice to share a validated model

Challenges:

- We used types (and dependent types) heavily.
  - e.g., indexed RTL values by bit size
  - e.g., indexed grammars, terms
- We used Coq's h.o. functions, notation for:
  - new grammar combinators
  - translation monad

# *Summary*

- A big part of formalization is modeling our environments.

- Mechanization makes it possible to scale beyond the toy models we used to do.
  - this is necessary to “widen” proofs to real code.

- But building models at scale is a big challenge.
  - validation & re-use are crucial
  - forces us to re-think how we do semantics
  - even old topics, like parsing, need to be revisited