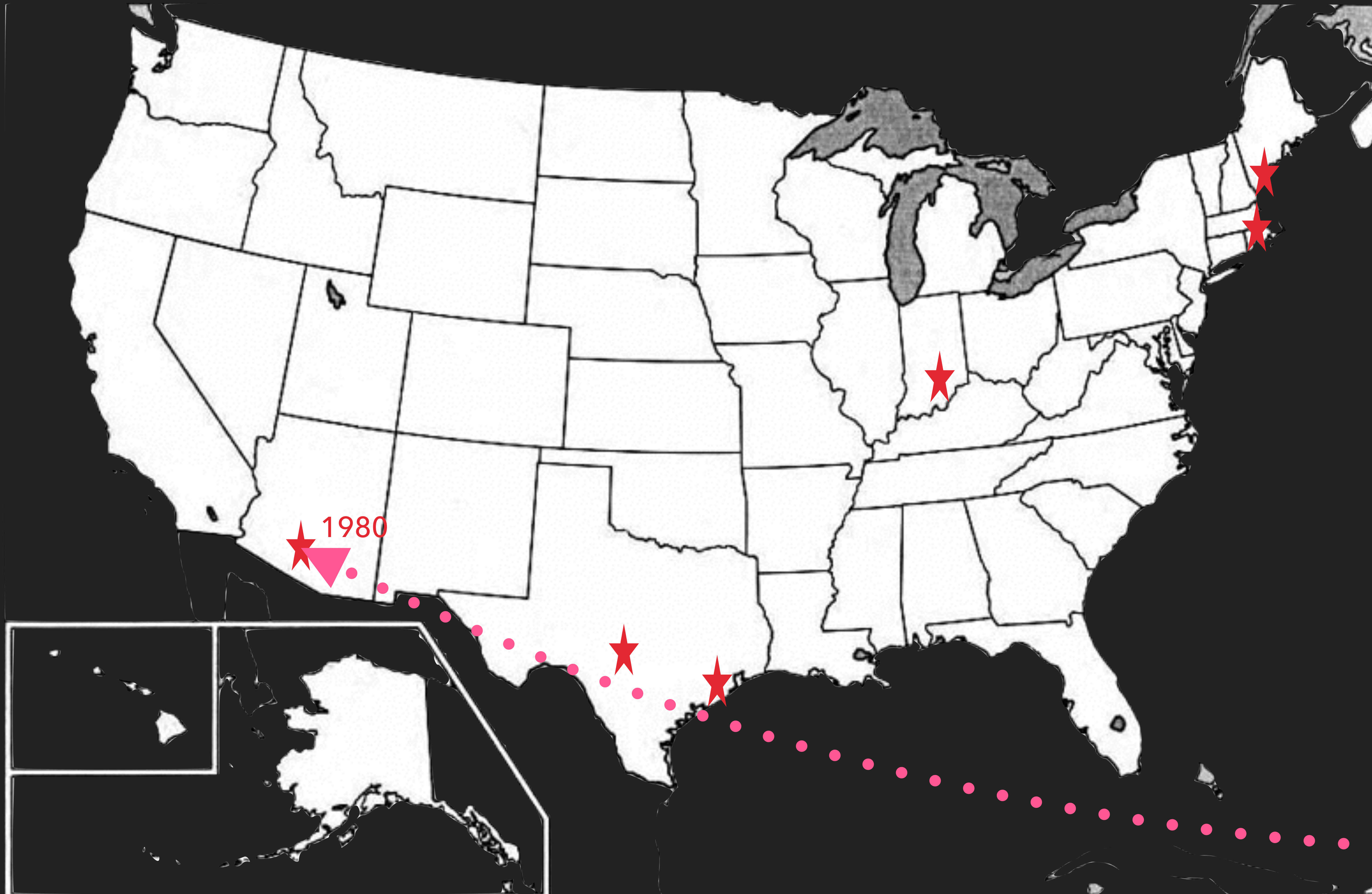
LOVE, MARRIAGE & HAPPINESS

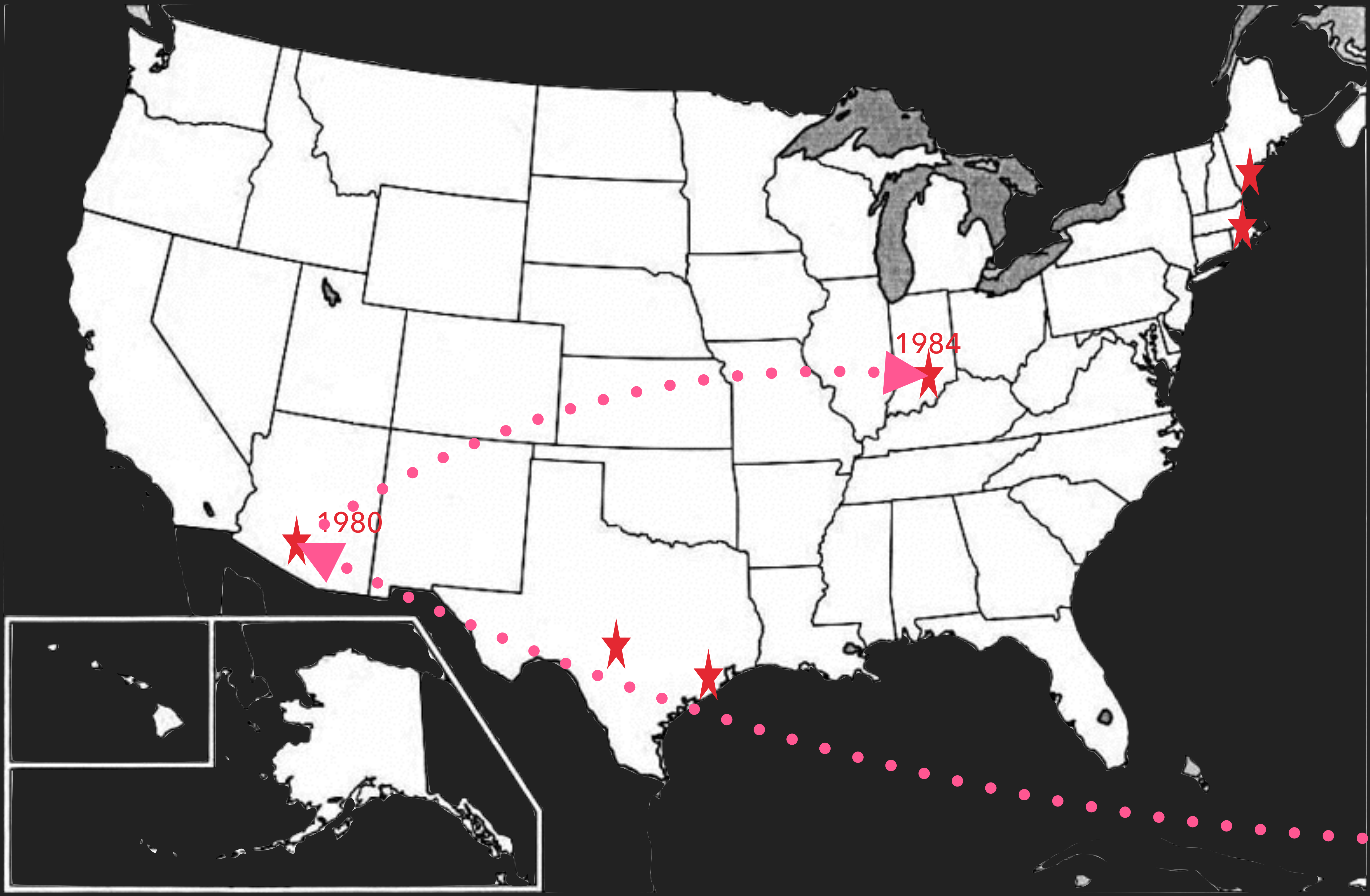
MATTHIAS FELLEISEN, NU PRL

MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)

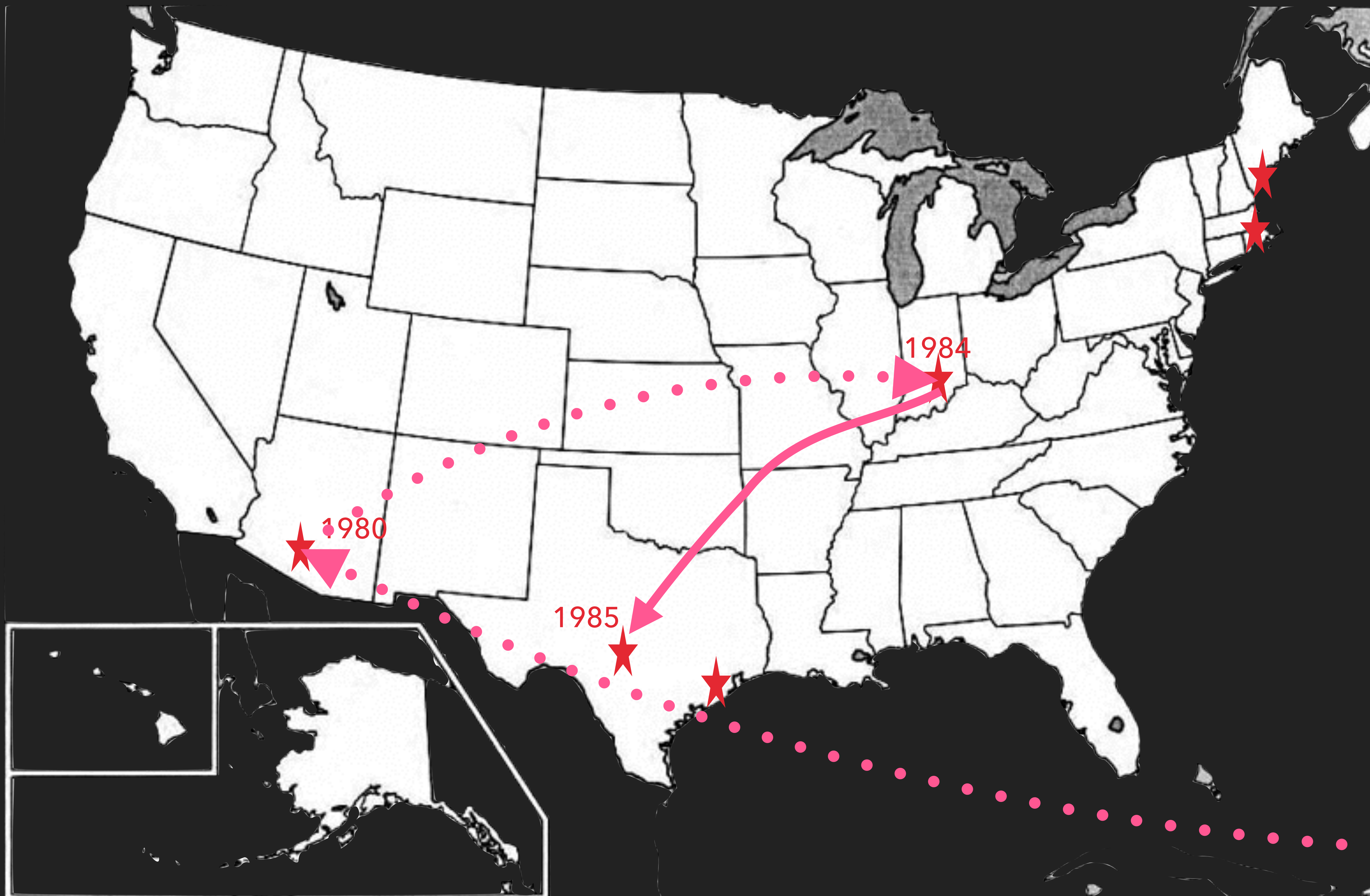
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



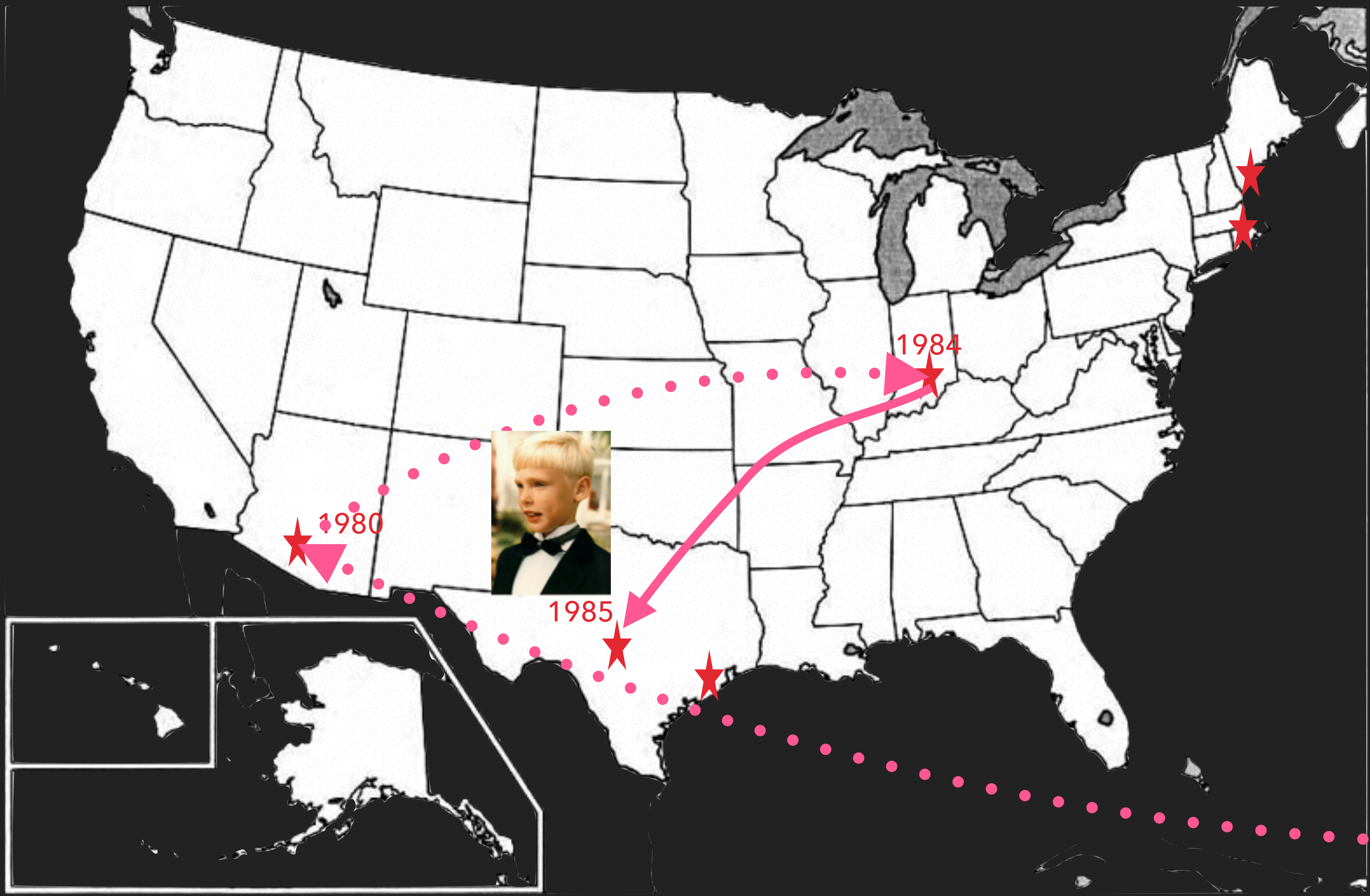
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



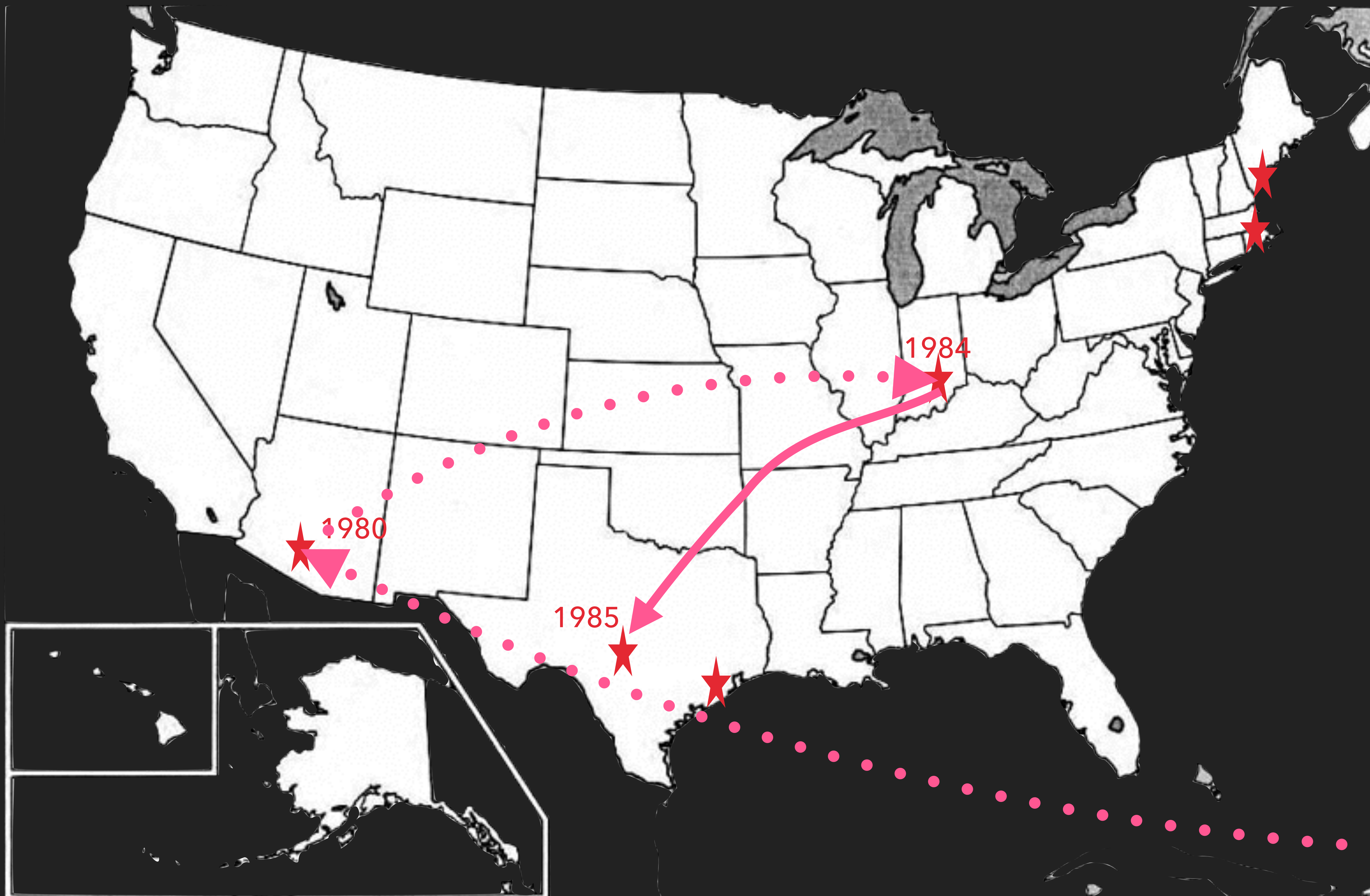
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



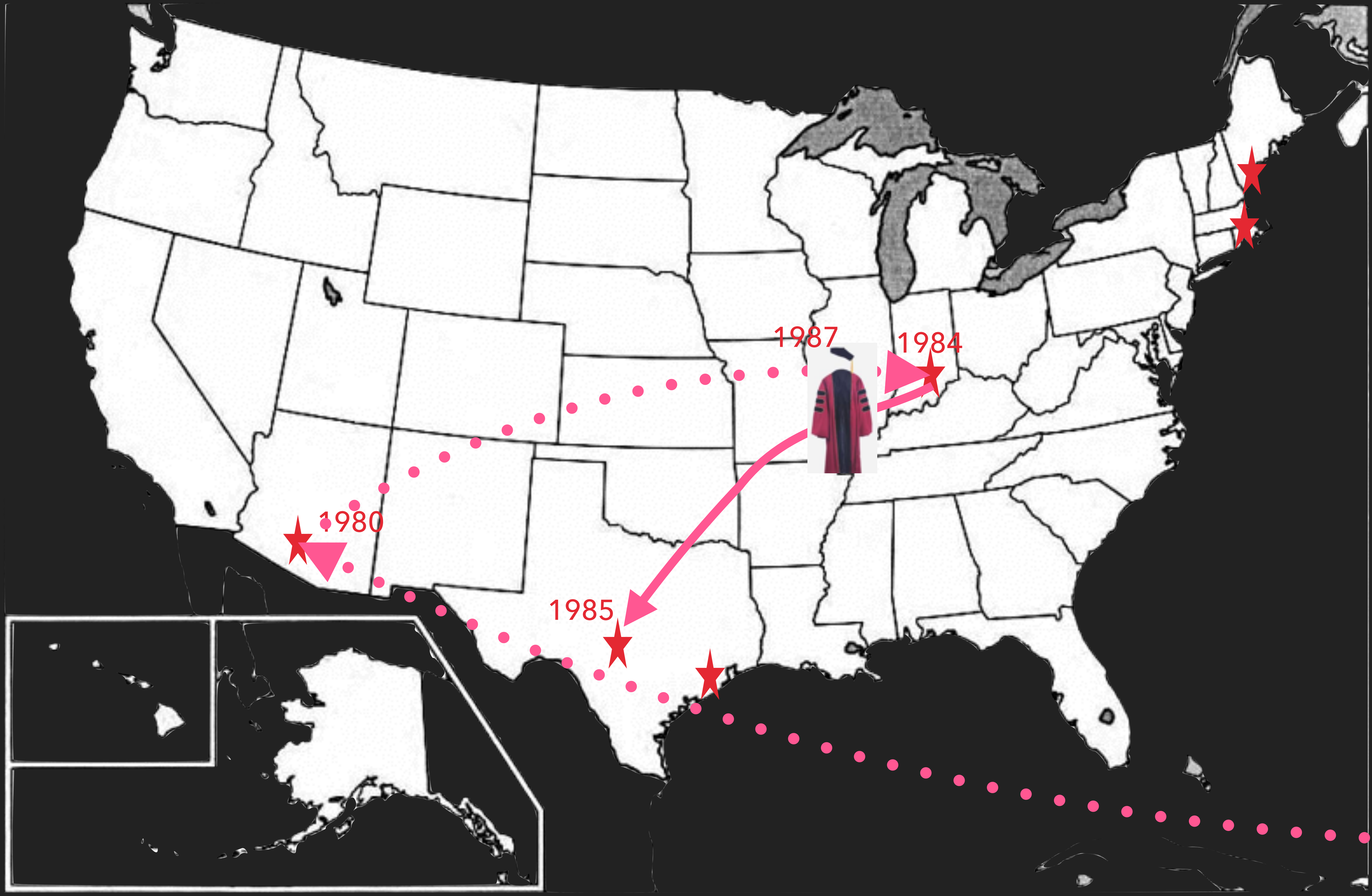
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



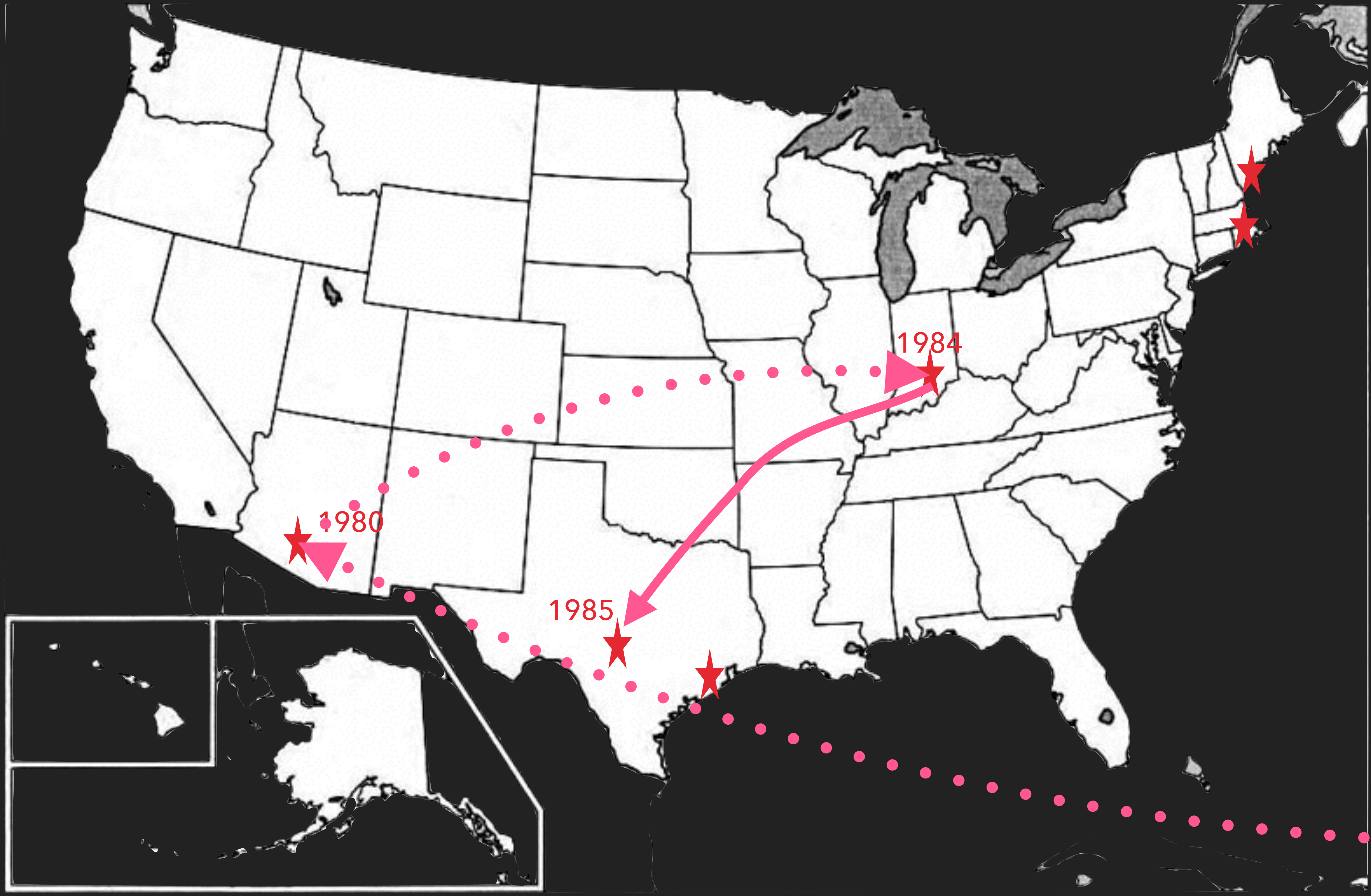
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



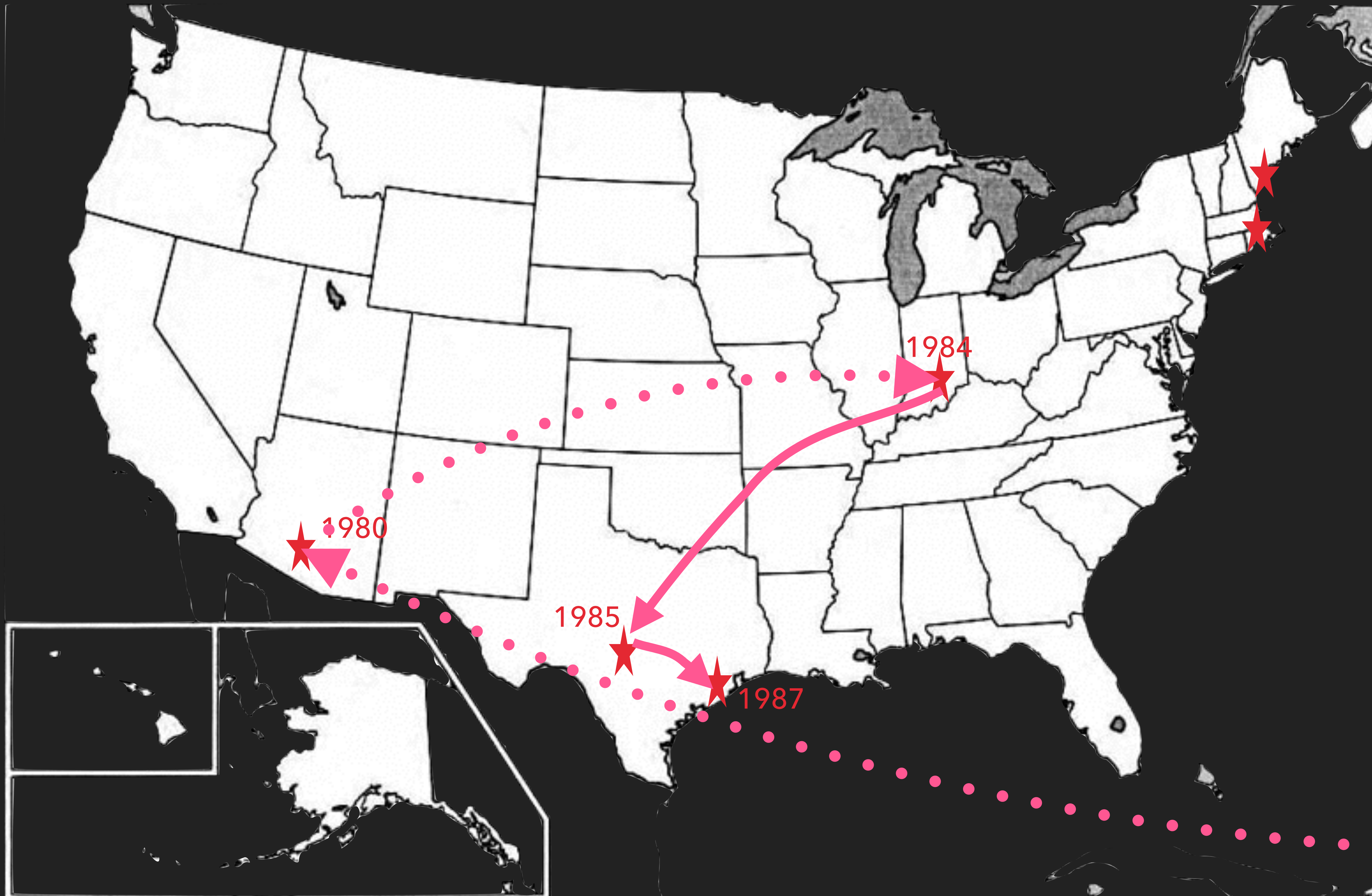
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



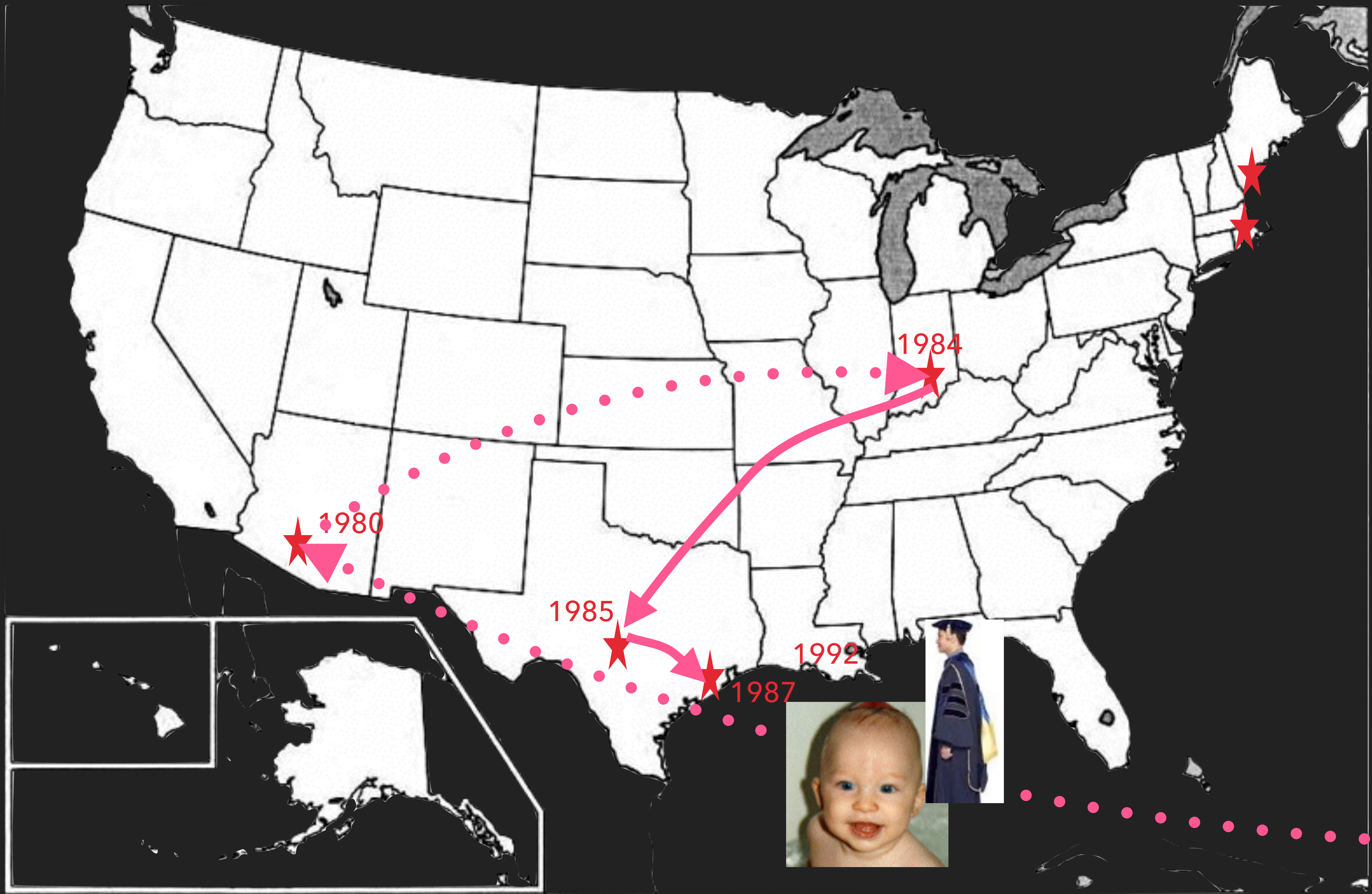
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



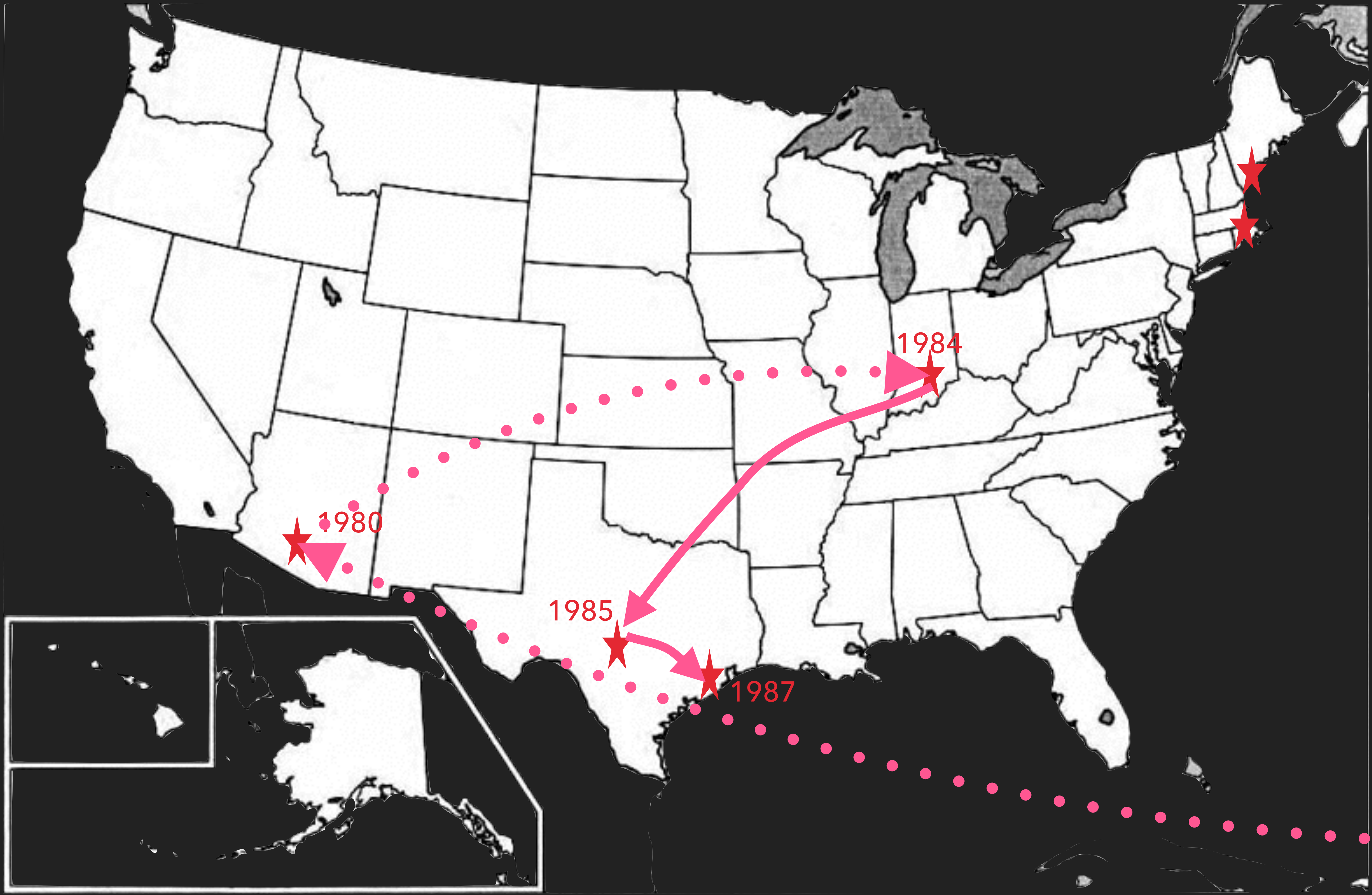
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



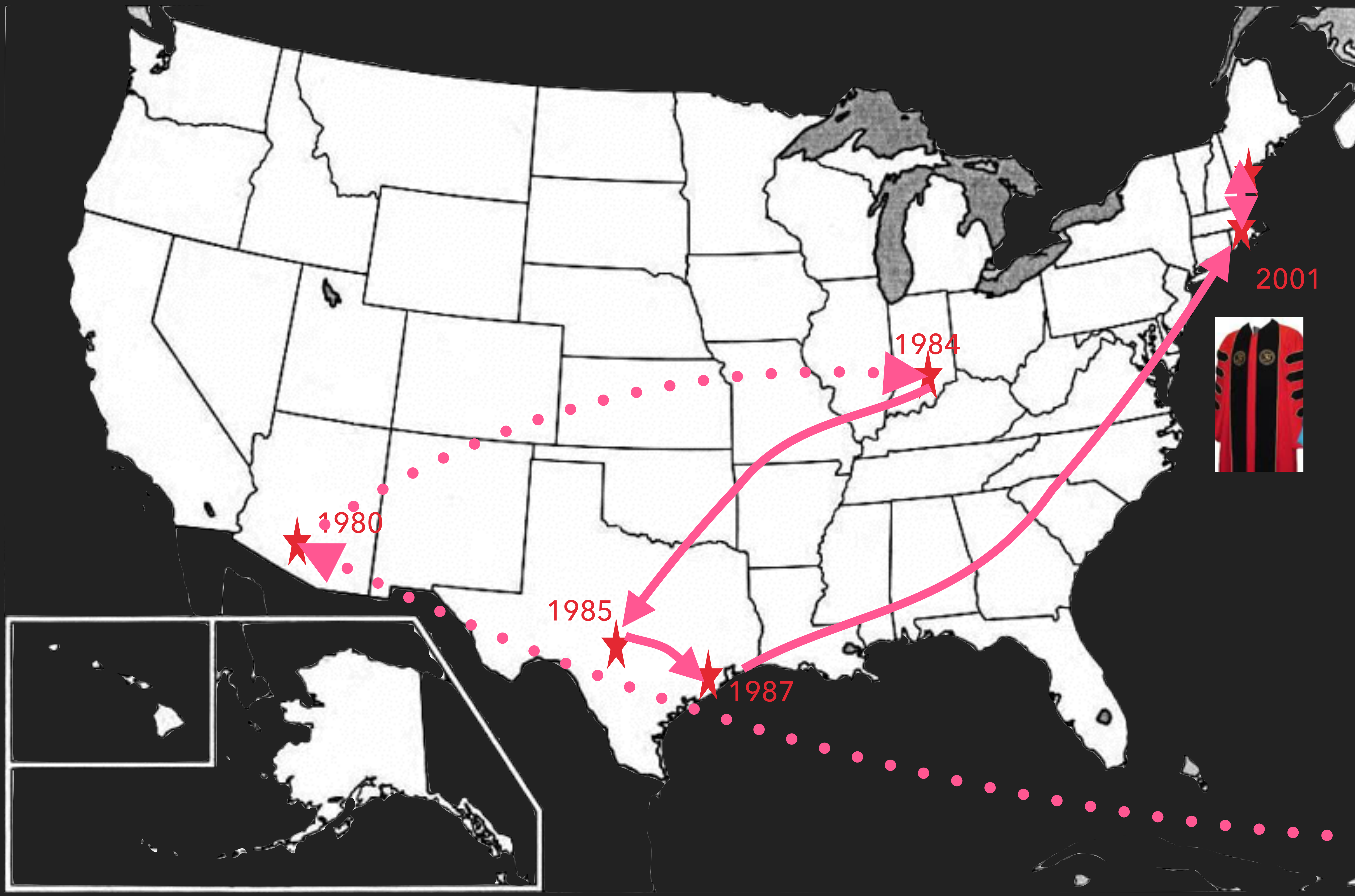
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



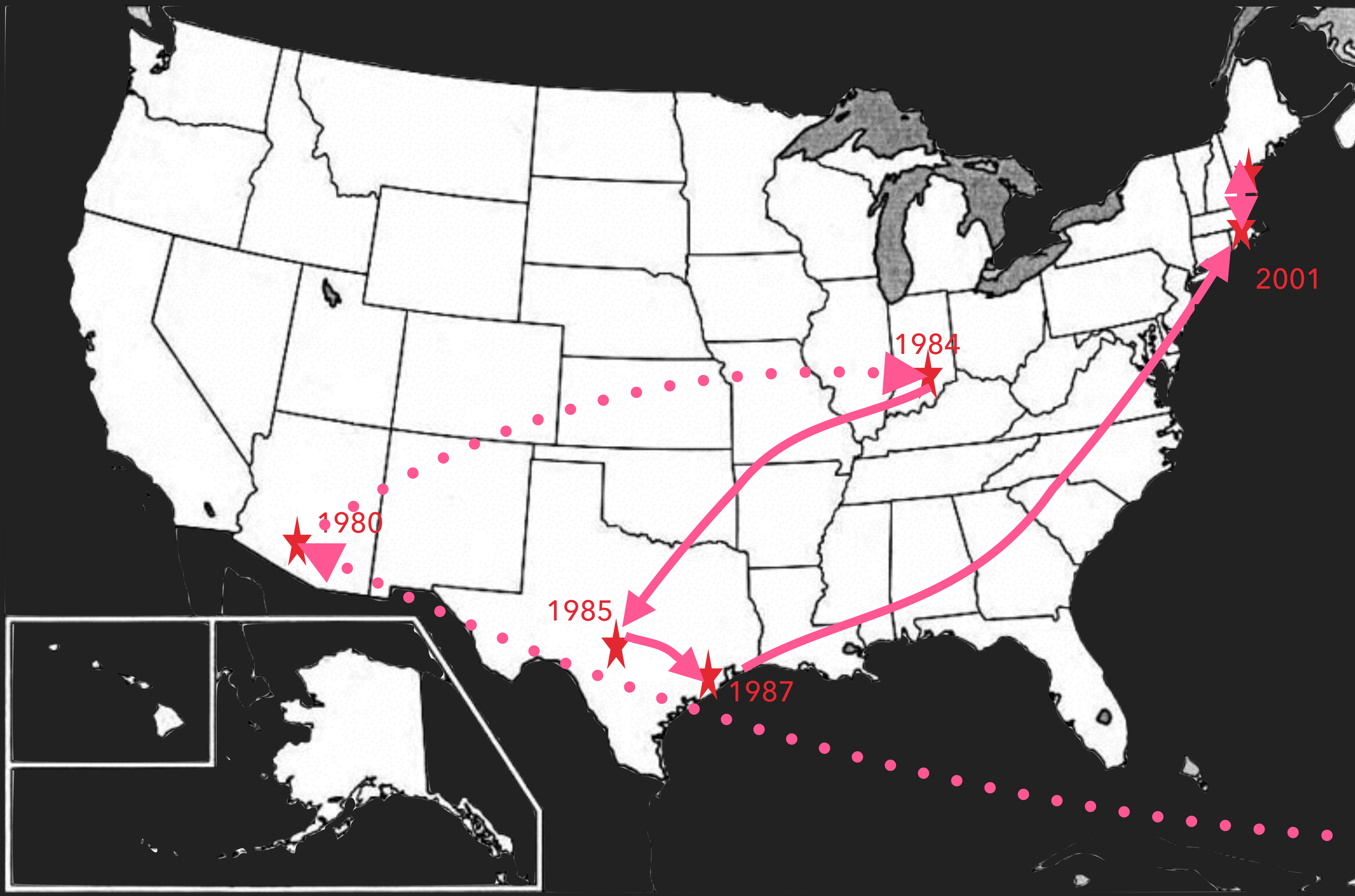
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



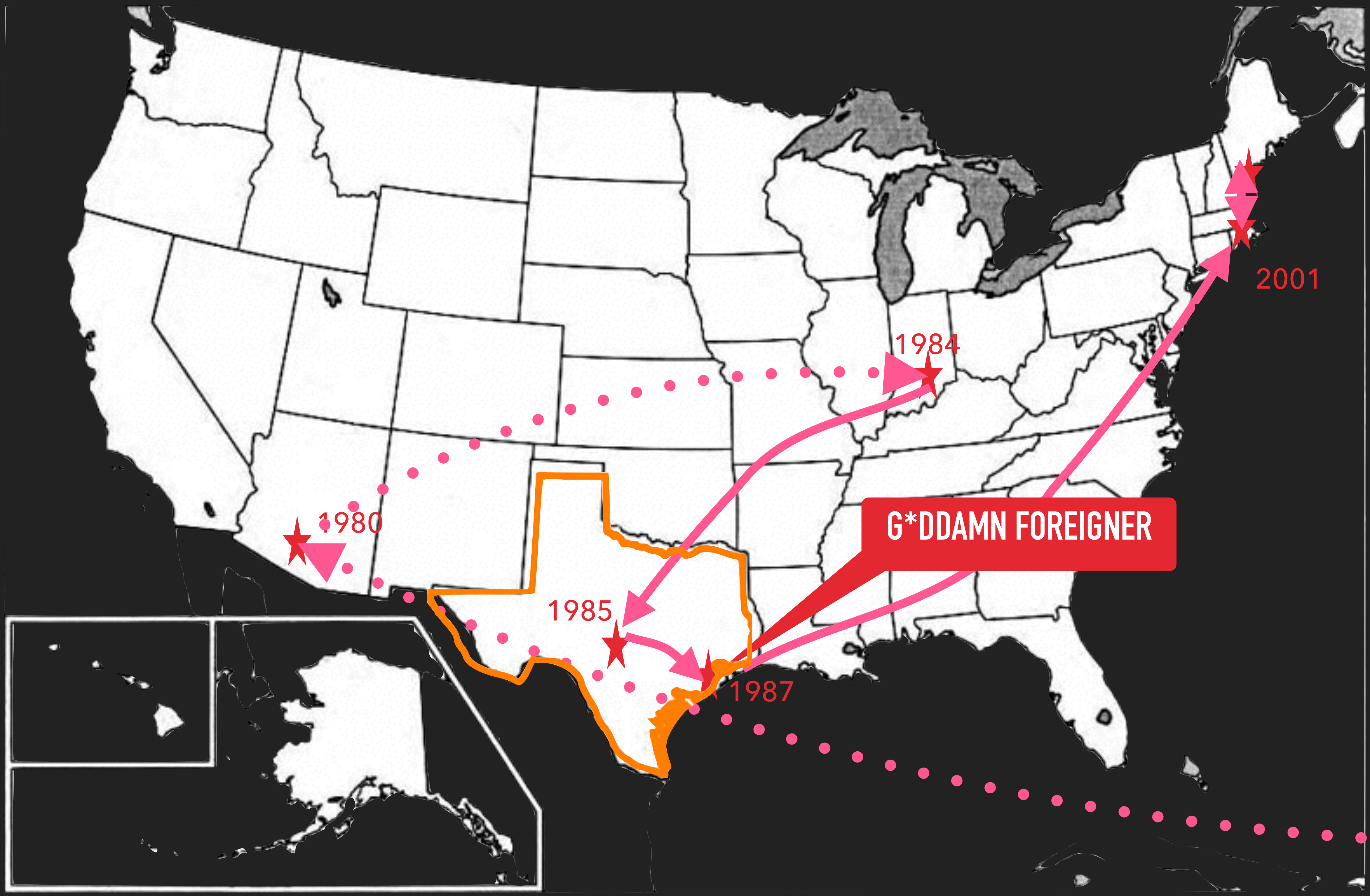
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



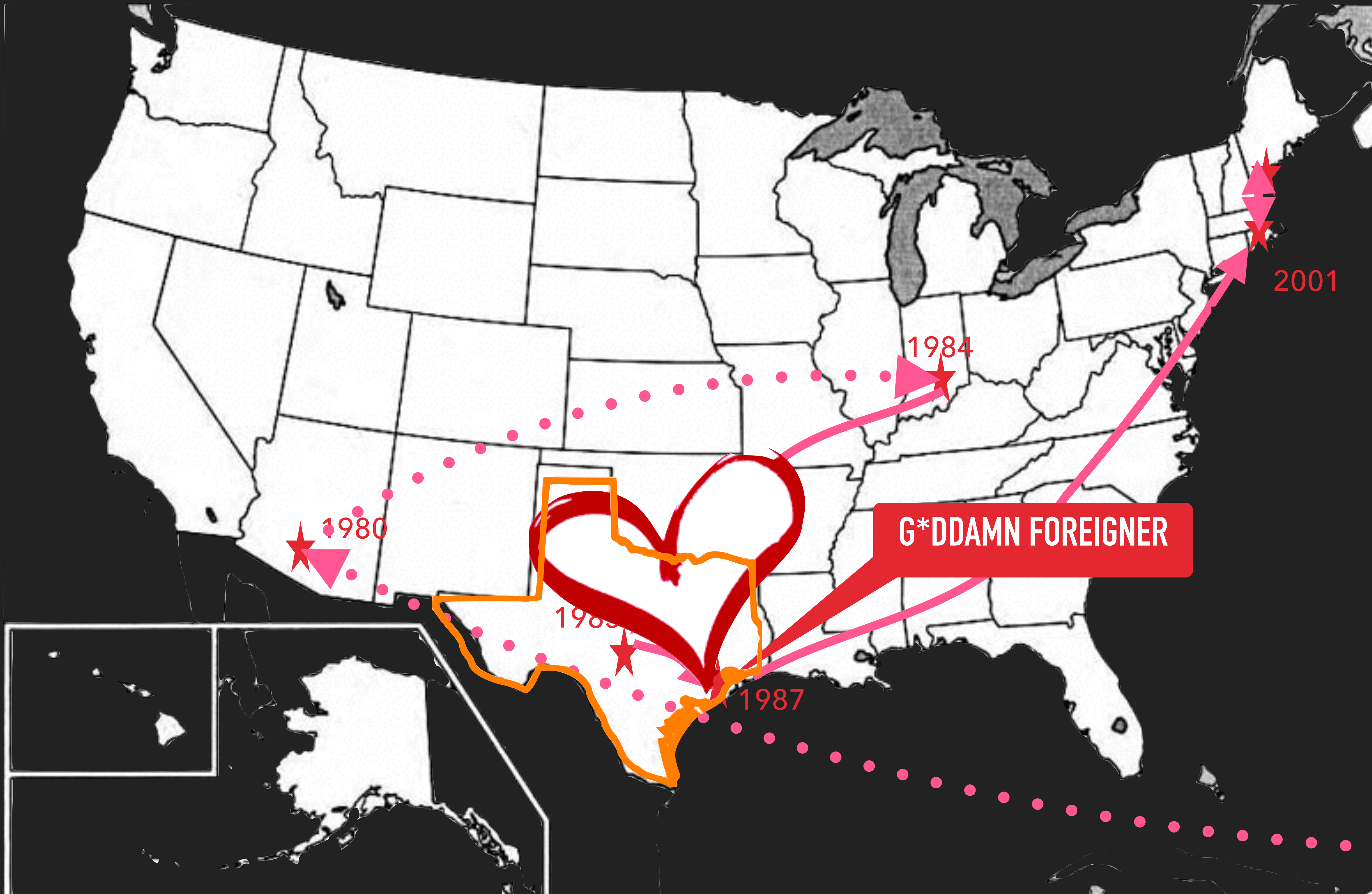
MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



MY CAREER, GEOGRAPHY & NUMBERS (THIS IS PLDI AFTER ALL)



WHAT'S LOVE GOT TO DO WITH IT

You are here because you fell in love with something about programming languages.

πrogramming

λanguages

YOU CAN'T JUST KEEP YOUR FINGERS OFF

πrogramming

λanguages

The most fundamental area of computer science. If you don't have a language, you can't compute.

YOU CAN'T JUST KEEP YOUR FINGERS OFF

Developers primarily use programming languages. The tools we build have meaning for them.

The most fundamental area of computer science. If you don't have a language, you can't compute.

пrogramming

λanguages

YOU CAN'T JUST KEEP YOUR FINGERS OFF

You will get to work with elegant mathematics and, some of you will develop new mathematics.

Developers primarily use programming languages. The tools we build have meaning for them.

The most fundamental area of computer science. If you don't have a language, you can't compute.

πrogramming

λanguages

YOU CAN'T JUST KEEP YOUR FINGERS OFF

Where else do you get to work with the coolest professors on the planet?

You will get to work with elegant mathematics and, some of you will develop new mathematics.

Developers primarily use programming languages. The tools we build have meaning for them.

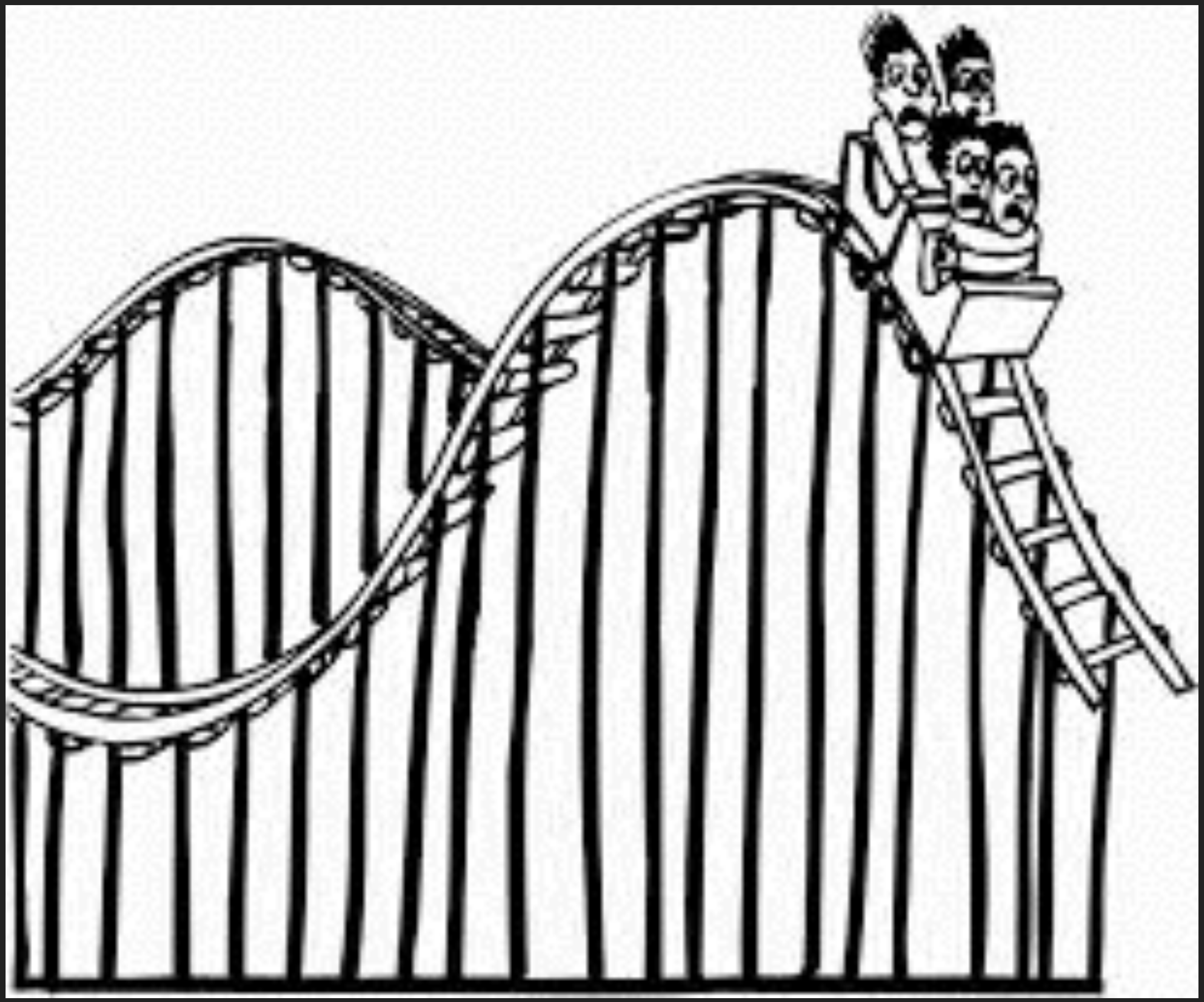
πrogramming

λanguages

The most fundamental area of computer science. If you don't have a language, you can't compute.

SO WHAT'S IT LIKE TO GET MARRIED TO PL RESEARCH

SO WHAT'S IT LIKE TO GET MARRIED TO PL RESEARCH



IT HAS ITS UPS AND DOWNS.

IT HAS ITS UPS AND DOWNS.

Falling in love.

IT HAS ITS UPS AND DOWNS.

Being in love.

Falling in love.

IT HAS ITS UPS AND DOWNS.

Getting thru difficult, troublesome times …

Being in love.

Falling in love.

AND LOVE IS WHAT GETS YOU BACK ON TRACK.

AND LOVE IS WHAT GETS YOU BACK ON TRACK.

But really, if you don't love PL, getting a PhD is hard.

AND LOVE IS WHAT GETS YOU BACK ON TRACK.

Hard because it's
an old and now
'hidden' discipline.

But really, if you don't love
PL, getting a PhD is hard.

AND LOVE IS WHAT GETS YOU BACK ON TRACK.

HARD because it isn't 'hot' with IT industry.

Hard because it's an old and now 'hidden' discipline.

But really, if you don't love PL, getting a PhD is hard.

AND LOVE IS WHAT GETS YOU BACK ON TRACK.

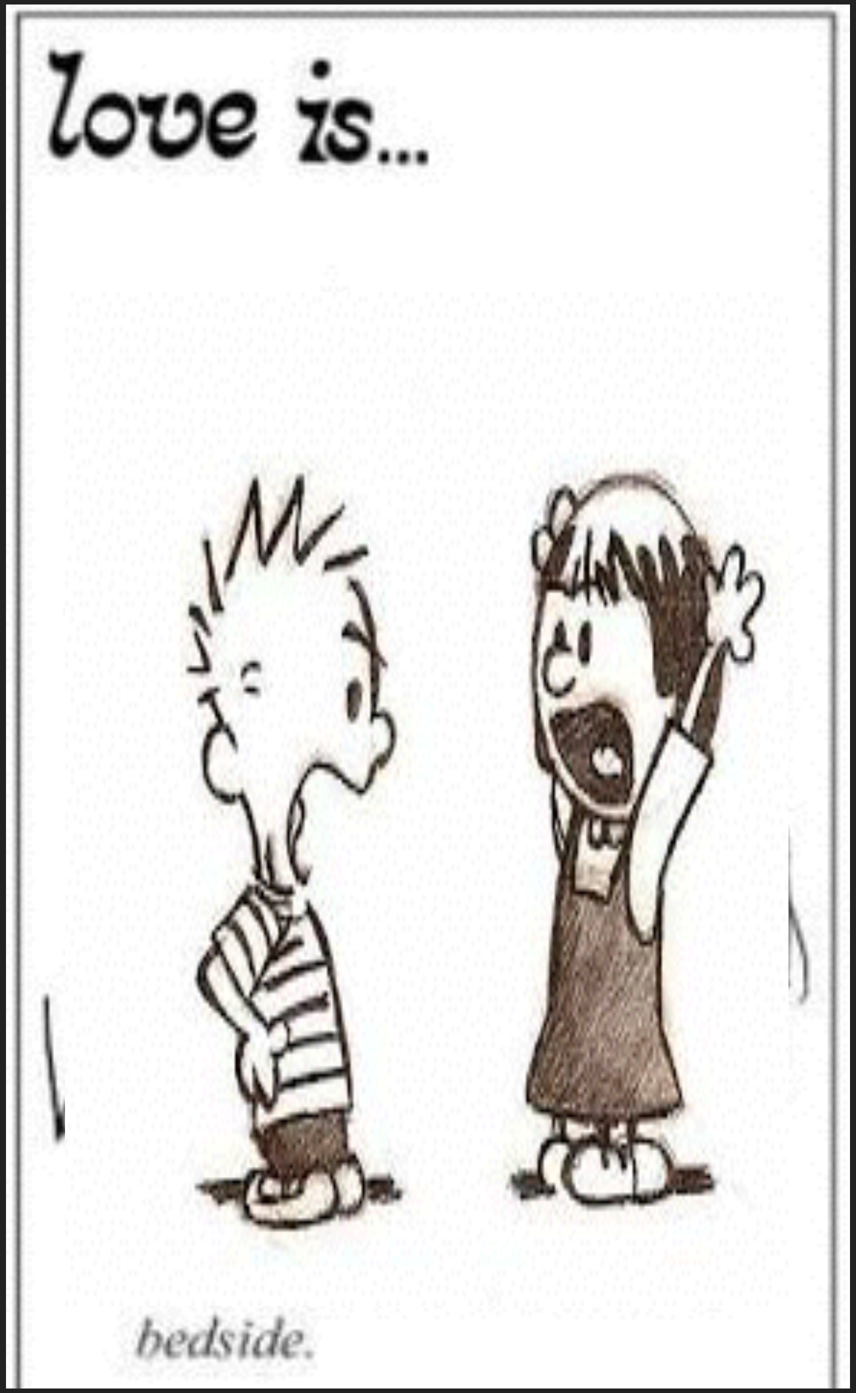
H.A.R.D.

HARD because it isn't 'hot' with IT industry.

Hard because it's an old and now 'hidden' discipline.

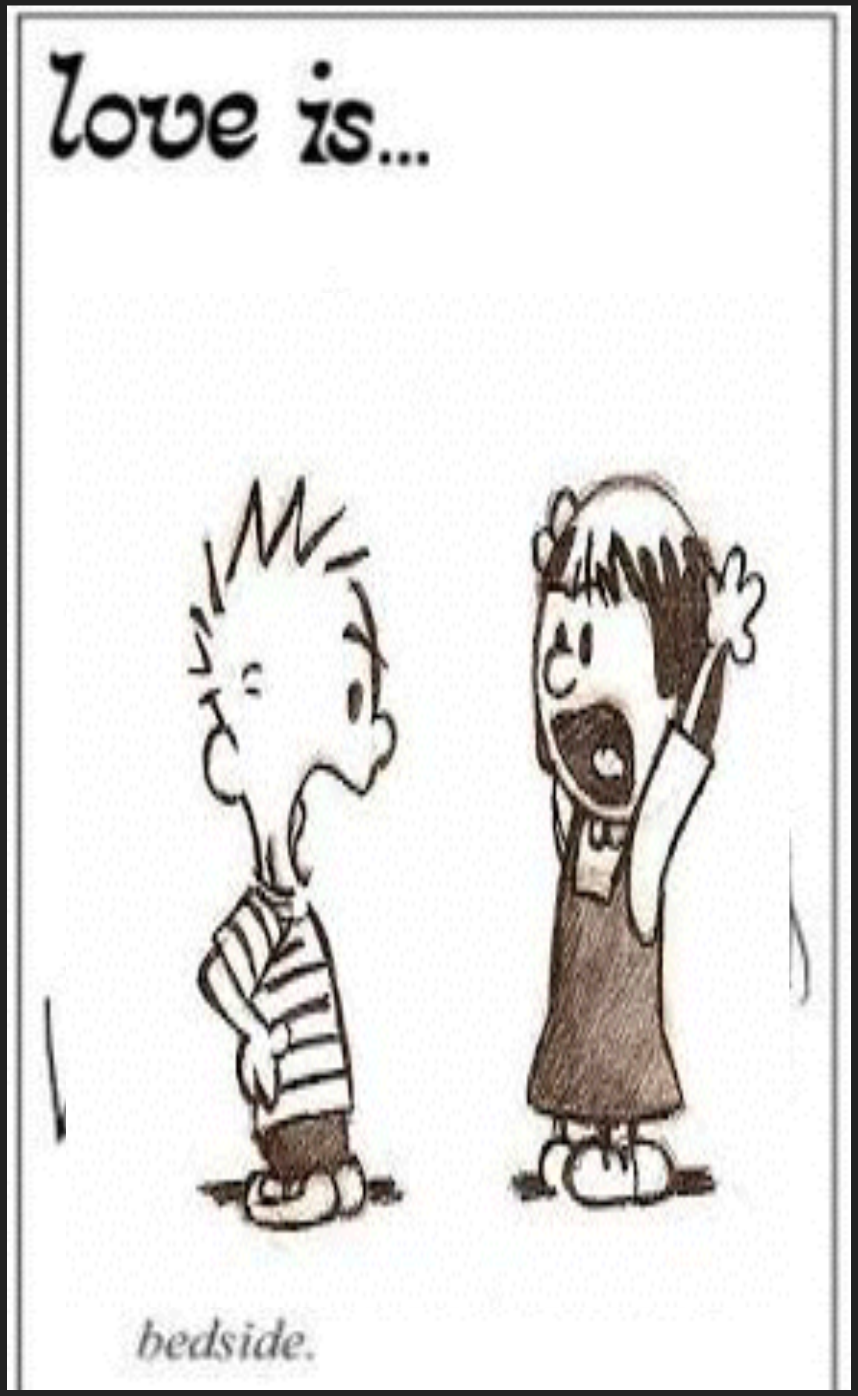
But really, if you don't love PL, getting a PhD is hard.

ONE LAST WARNING: IF YOU DON'T LOVE IT, LEAVE IT.



ONE LAST WARNING: IF YOU DON'T LOVE IT, LEAVE IT.

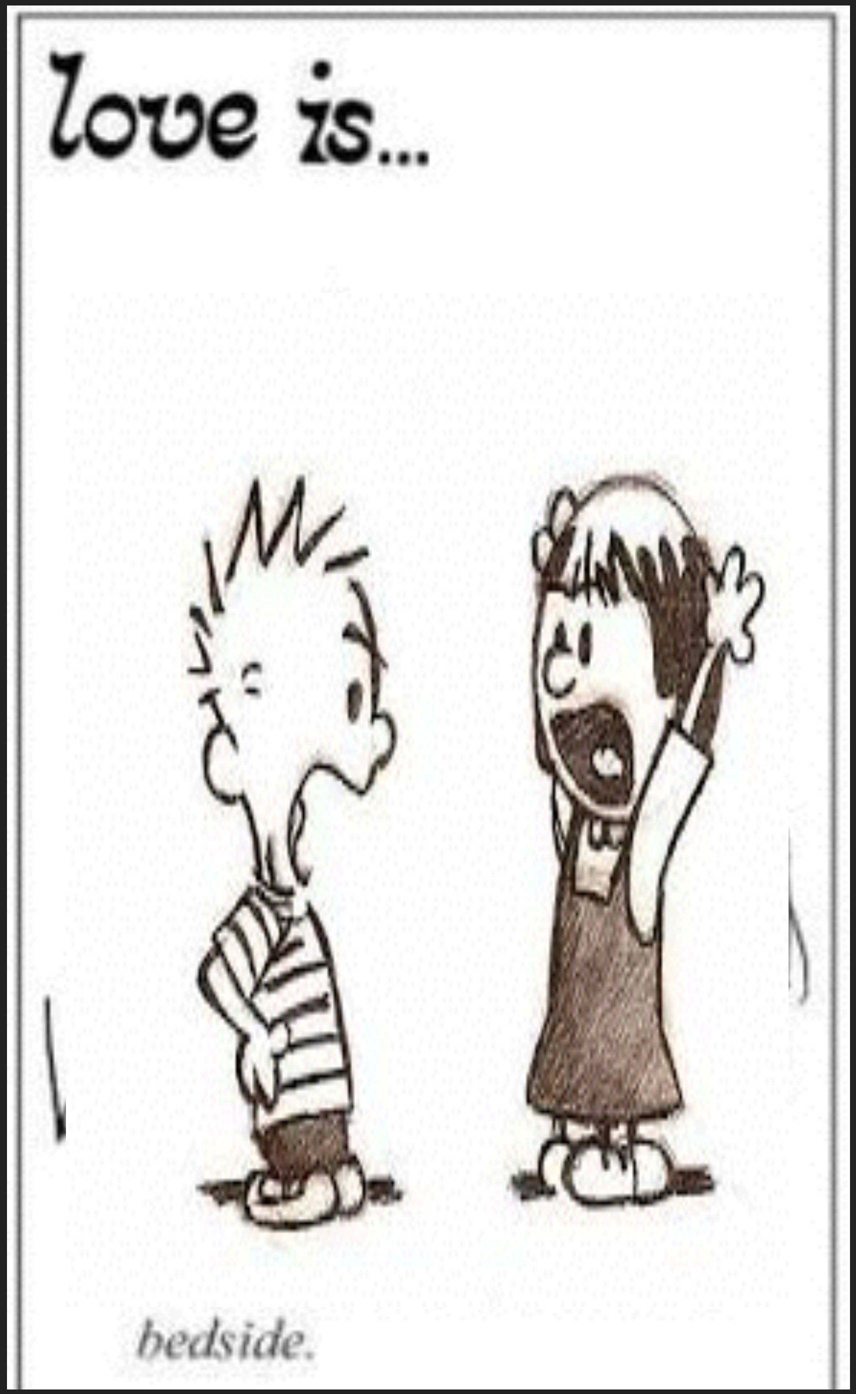
If you want to be famous, get into *Artificial Intelligence*.



ONE LAST WARNING: IF YOU DON'T LOVE IT, LEAVE IT.

If you want to make money, do *Big Data*.

If you want to be famous, get into *Artificial Intelligence*.

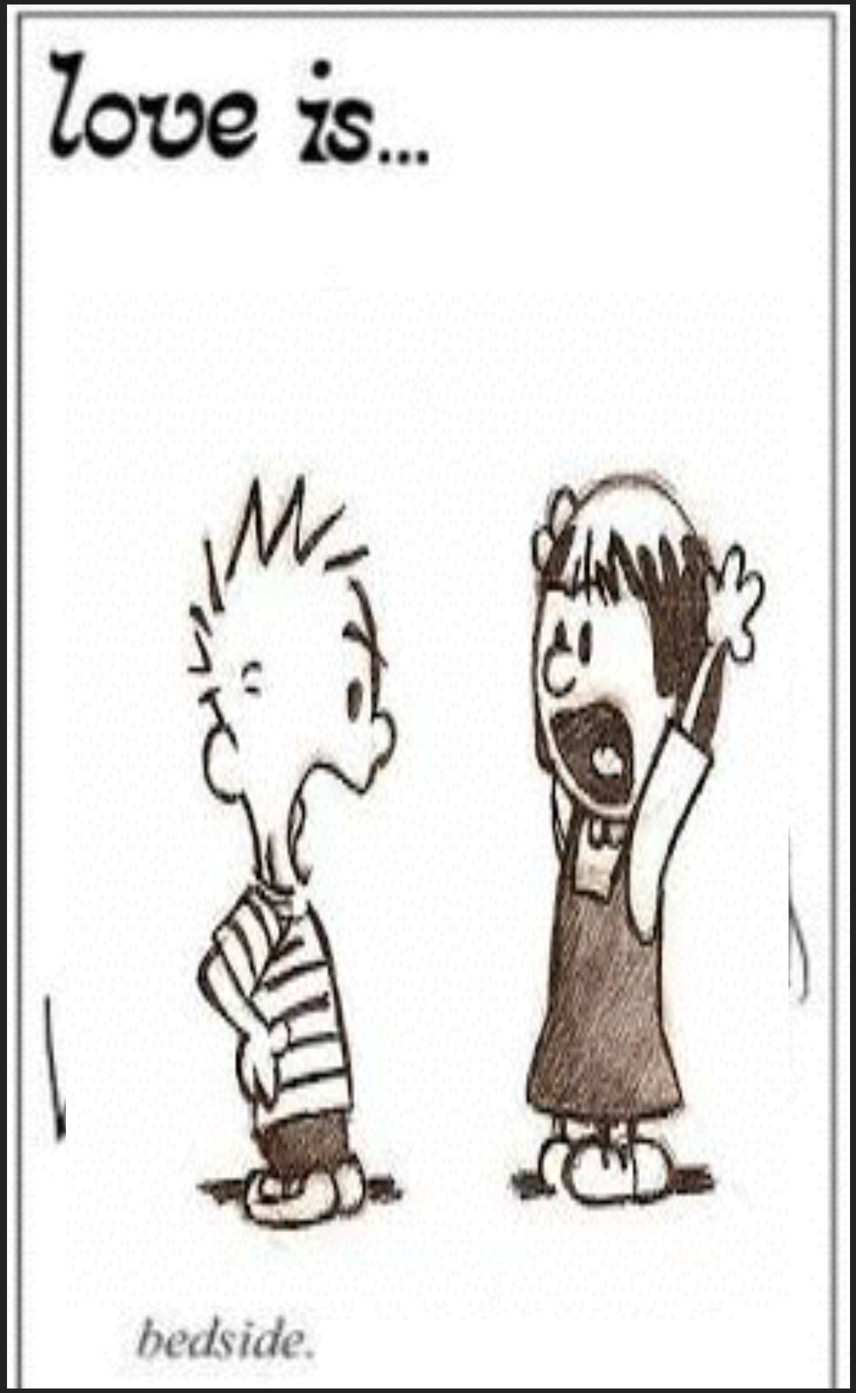


ONE LAST WARNING: IF YOU DON'T LOVE IT, LEAVE IT.

If you want a career, switch majors. I hear our *Business School* is looking for students.

If you want to make money, do *Big Data*.

If you want to be famous, get into *Artificial Intelligence*.



TYPES FOR UNTYPED LANGUAGES, HOW LOVE WORKS

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

CORKY CARTWRIGHT
"LET'S WORK ON TYPES FOR SCHEME"

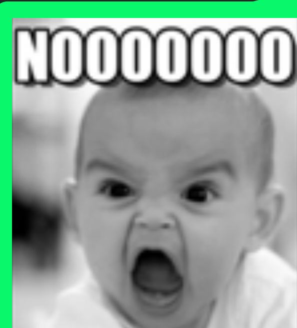
THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

CORKY CARTWRIGHT
“LET’S WORK ON TYPES
FOR SCHEME”

(It doesn’t have to be
love at first sight.)

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

CORKY CARTWRIGHT
“LET’S WORK ON TYPES FOR SCHEME”



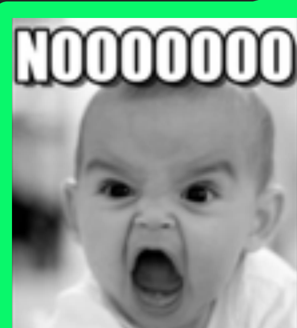
COOL! WORKING WITH CARTWRIGHT AND FAGAN

(It doesn’t have to be love at first sight.)

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE ONLY USER OF ANDREW WRIGHT'S SOFT SCHEME

CORKY CARTWRIGHT "LET'S WORK ON TYPES FOR SCHEME"



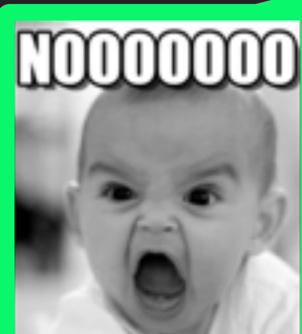
(It doesn't have to be love at first sight.)

COOL! WORKING WITH CARTWRIGHT AND FAGAN

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE ONLY USER OF ANDREW WRIGHT'S SOFT SCHEME

CORKY CARTWRIGHT "LET'S WORK ON TYPES FOR SCHEME"



(It doesn't have to be love at first sight.)

COOL! WORKING WITH CARTWRIGHT AND FAGAN

CO-CREATED SPIDEY SCHEME WITH CORMAC FLANAGAN

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

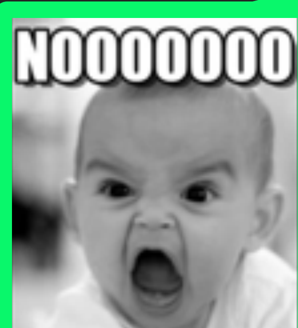
THE ONLY USER OF ANDREW WRIGHT'S SOFT SCHEME

CORKY CARTWRIGHT "LET'S WORK ON TYPES FOR SCHEME"

MODULAR SPIDER WITH PHILIPPE MEUNIER

CO-CREATED SPIDEY SCHEME WITH CORMAC FLANAGAN

COOL! WORKING WITH CARTWRIGHT AND FAGAN

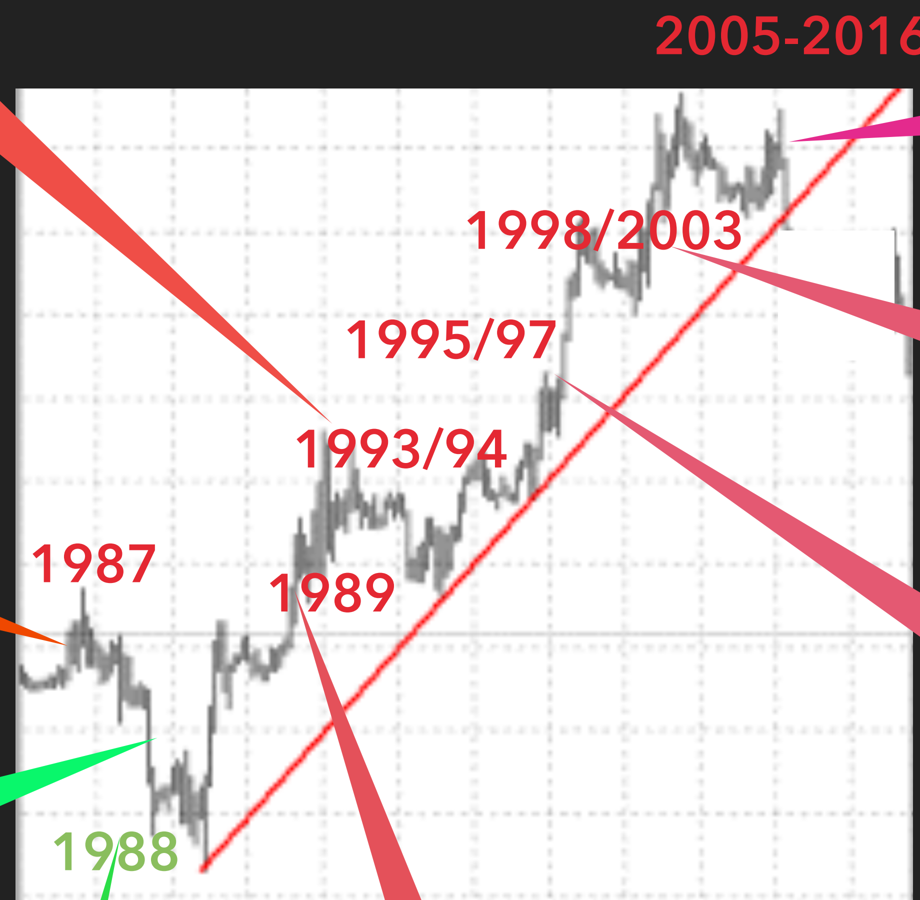


(It doesn't have to be love at first sight.)

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE ONLY USER OF ANDREW WRIGHT'S SOFT SCHEME

TYPED RACKET W/ TOBIN-HOCHSTADT



CORKY CARTWRIGHT "LET'S WORK ON TYPES FOR SCHEME"

MODULAR SPIDER WITH PHILIPPE MEUNIER

CO-CREATED SPIDEY SCHEME WITH CORMAC FLANAGAN

COOL! WORKING WITH CARTWRIGHT AND FAGAN

(It doesn't have to be love at first sight.)

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (bond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

1987

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (bond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

1987

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (bond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

THERE IS LOTS OF LISP OUT THERE
AND THEY MAY WANT TYPES.

1987

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (bond
                 ? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

WE WANT WIDE-SPECTRUM PROGRAMMING.

THERE IS LOTS OF LISP OUT THERE AND THEY MAY WANT TYPES.

1987

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

PROGRAMMERS DO NOT WANT TO COPE WITH THE IDIOSYNCRASIES OF TYPE SYSTEMS.

```
(bond
        ? p) p]
 [e  e  (and (tautology? (p true)) (tautology? (p false)))]))
```

WE WANT WIDE-SPECTRUM PROGRAMMING.

THERE IS LOTS OF LISP OUT THERE AND THEY MAY WANT TYPES.

1987

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

IT'LL COME TRUE IN 10 OR 20 YEARS.

PROGRAMMERS DO NOT WANT TO COPE WITH THE IDIOSYNCRASIES OF TYPE SYSTEMS.

```
(bond
          p) p]
   [else (and (tautology? (p true)) (tautology? (p false)))]))
```

WE WANT WIDE-SPECTRUM PROGRAMMING.

THERE IS LOTS OF LISP OUT THERE AND THEY MAY WANT TYPES.

1987

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

IT'LL COME TRUE IN 10 OR 20 YEARS.

PROGRAMMERS DO NOT WANT TO COPE WITH THE IDIOSYNCRASIES OF TYPE SYSTEMS.

```
(bond
       p) p]
  [else (and (tautology? (p true)) (tautology? (p false)))]))
```

WE WANT WIDE-SPECTRUM PROGRAMMING.

THERE IS LOTS OF LISP OUT THERE AND THEY MAY WANT TYPES.

1987

JavaScript

php

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

WHAT'S THE TYPE OF AN UNTYPED PROGRAM?

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

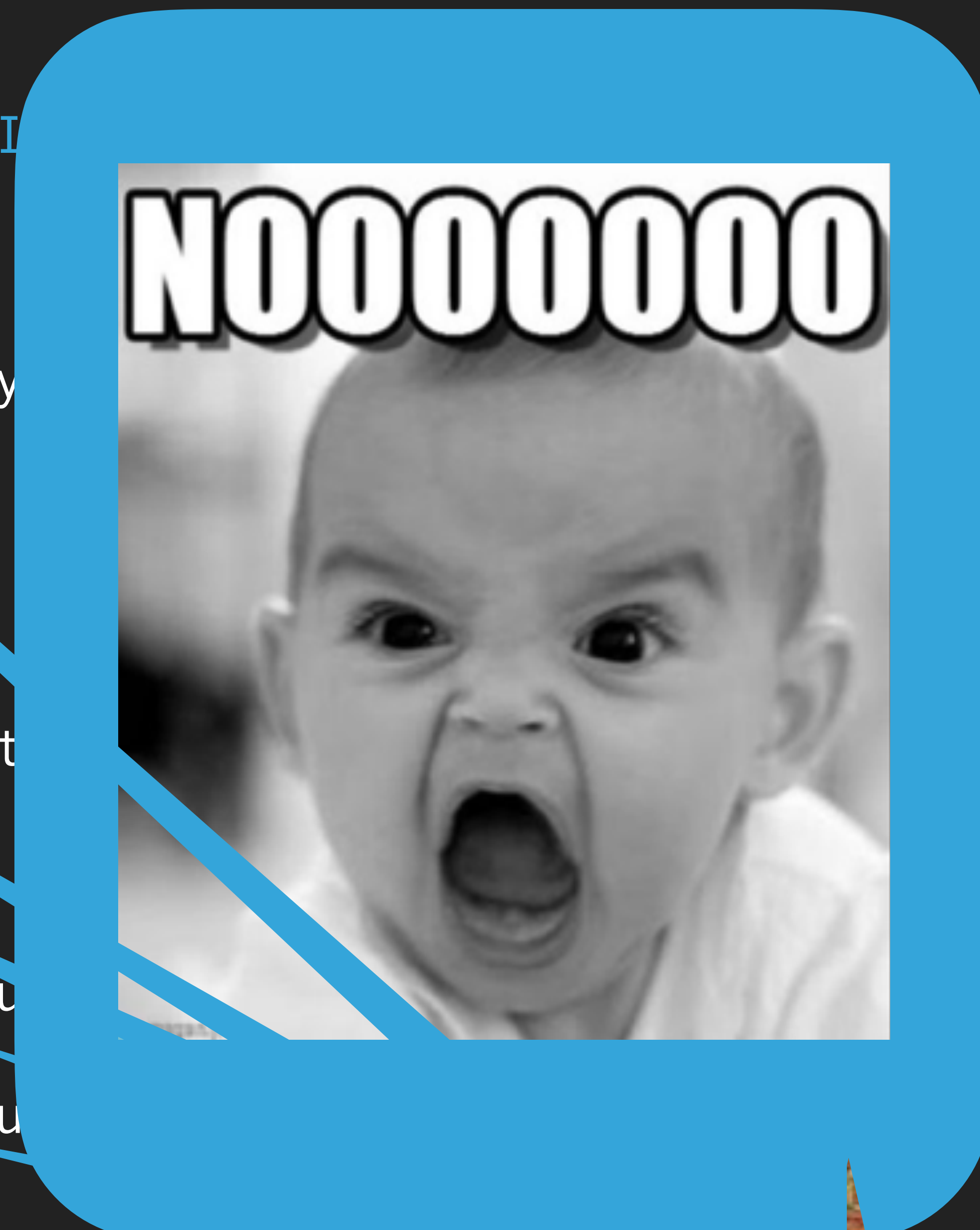
EASY! TYPE INFERENCE! ML HAS HAD IT SINCE 1978.

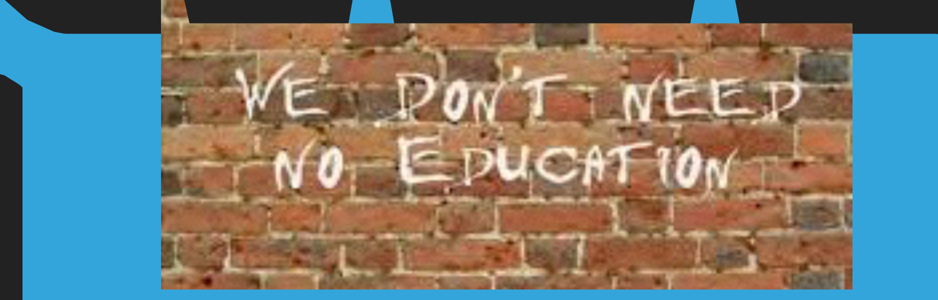
WE CAN SAY IT IN OCAML

```
type proposition = InL of bool | InR of (bool -> proposition)

let rec is_tautology p =
 match p with
  | InL b -> b
  | InR p -> is_tautology(p true) && is_tautology(p false)

is_tautology (InR(fun x -> InL true))
is_tautology (InR(fun x -> InR(fun y -> or then InL x else InL y)))
```

WE CAN SAY IT IN OCAML

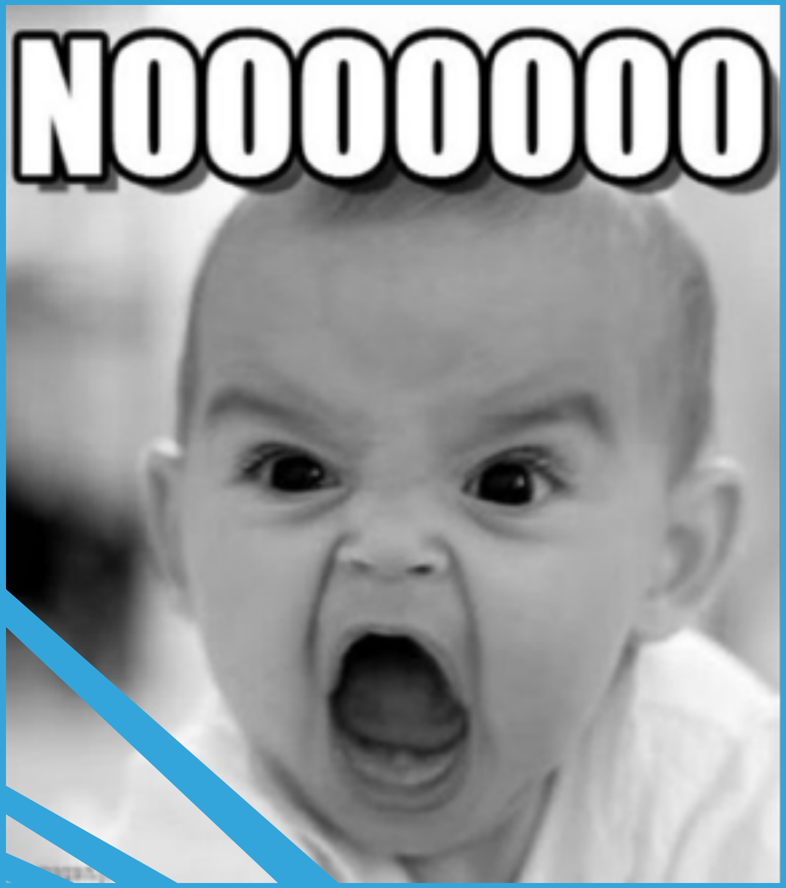
```
type proposition = I                          roposition)

let rec is_tautology

 match p with

   | InL b -> b

   | InR p -> is_taut                         o false)


is_tautology (InR(fu

is_tautology (InR(fu                          L x else InL y)))
```





WE CAN SAY IT IN OCAML

```
type proposition = I                              roposition)

let rec is_tautology

 match p with

   | InL b -> b

   | InR p -> is_taut                              p false);;

is_tautology (InR(fu

is_tautology (InR(fu                               L x else InL y)))
```



WE DON'T WANT TO WRITE DOWN TYPE DEFINITIONS. WE DON'T WANT TO ADD INSERTIONS AND PROJECTIONS.

SO HOW COULD WE SAY THIS IN SCHEME?


SOFT TYPING

Robert Cartwright, Mike Fagan*
Department of Computer Science
Rice University
Houston, TX 77251-1892

- replace ML's type algebra (x, *, ->, …)
- with Remy's extensible records exclusively
- make it work for 100-line purely functional programs in quasi-Scheme

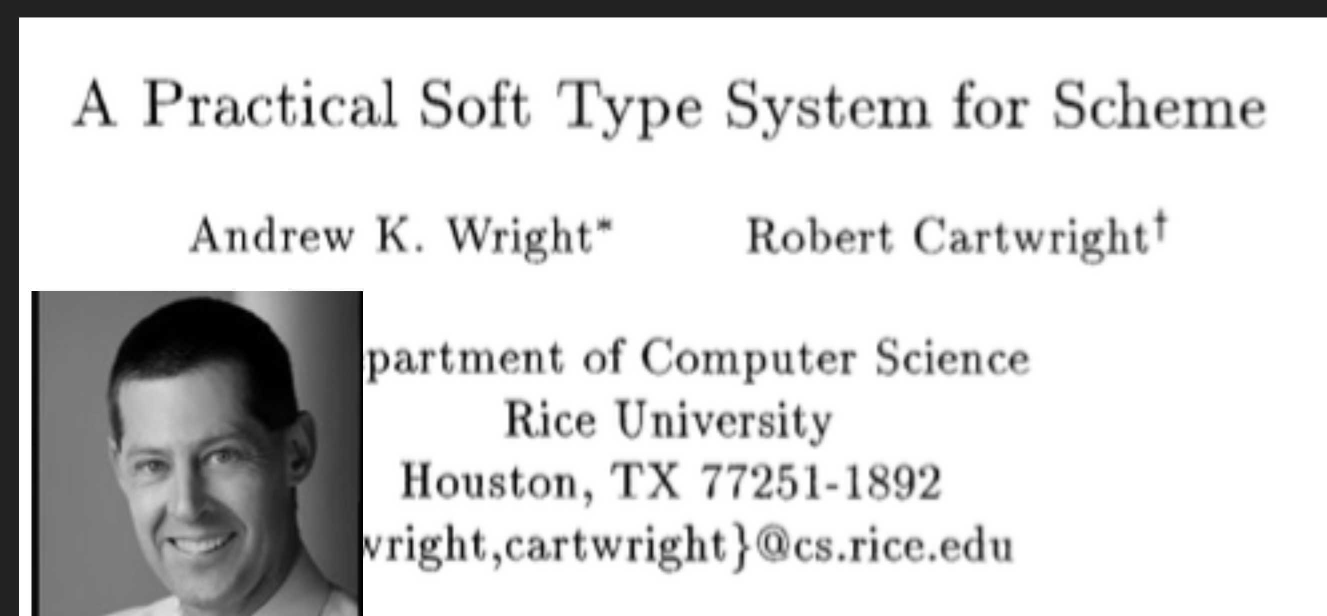
SO HOW COULD WE SAY THIS IN SCHEME?

SOFT TYPING

Robert Cartwright, Mike Fagan*
Department of Computer Science
Rice University
Houston, TX 77251-1892

- replace ML's type algebra (x, *, ->, …)
- with Remy's extensible records exclusively
- make it work for 100-line purely functional programs in quasi-Scheme

- grow it to full Chez Scheme
- whole-program inference
- success: speed-up



A Practical Soft Type System for Scheme

Andrew K. Wright*   Robert Cartwright†

partment of Computer Science
Rice University
Houston, TX 77251-1892
vright,cartwright}@cs.rice.edu

WHAT SOFT SCHEME CAN DO

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

infer via modified HM

```
(-> (μ (Proposition)
       (+ Boolean (-> Boolean Proposition)))
    Boolean)
```

WHAT SOFT SCHEME CAN DO

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

infer via modified HM

```
(-> (μ (Proposition)
       (+ Boolean (-> Boolean Proposition)))
    Boolean)
```

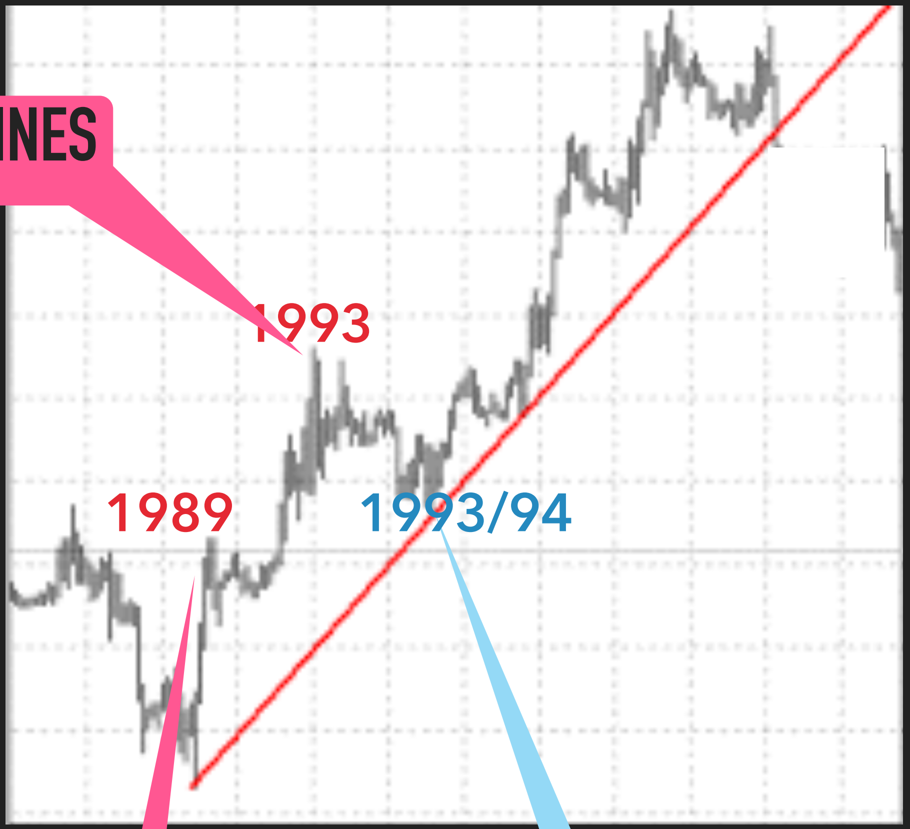
THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

WRIGHT CAN CHECK 1,000 LINES

FAGAN CAN CHECK 100 LINES.

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

WRIGHT CAN CHECK 1,000 LINES



FAGAN CAN CHECK 100 LINES.

YOURS TRULY STRUGGLES WITH DOZENS OF SMALL AND LARGE PROGRAMS.

WHAT SOFT SCHEME CAN'T DO.


```
(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (p false))]))
```

WHAT SOFT SCHEME CAN'T DO.


```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (p false))]))
```

DOZENS OF LINES FOR THE TYPE MISMATCH W/O TELLING THE DEV WHERE THINGS WENT WRONG

WHAT SOFT SCHEME CAN'T DO.


```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (p false))]))
```

formulate

DOZENS OF LINES FOR THE TYPE MISMATCH W/O TELLING THE DEV WHERE THINGS WENT WRONG

```
any sensible type-error
message,

just one, please .. .. ..
```

WHAT SOFT SCHEME CAN'T DO.

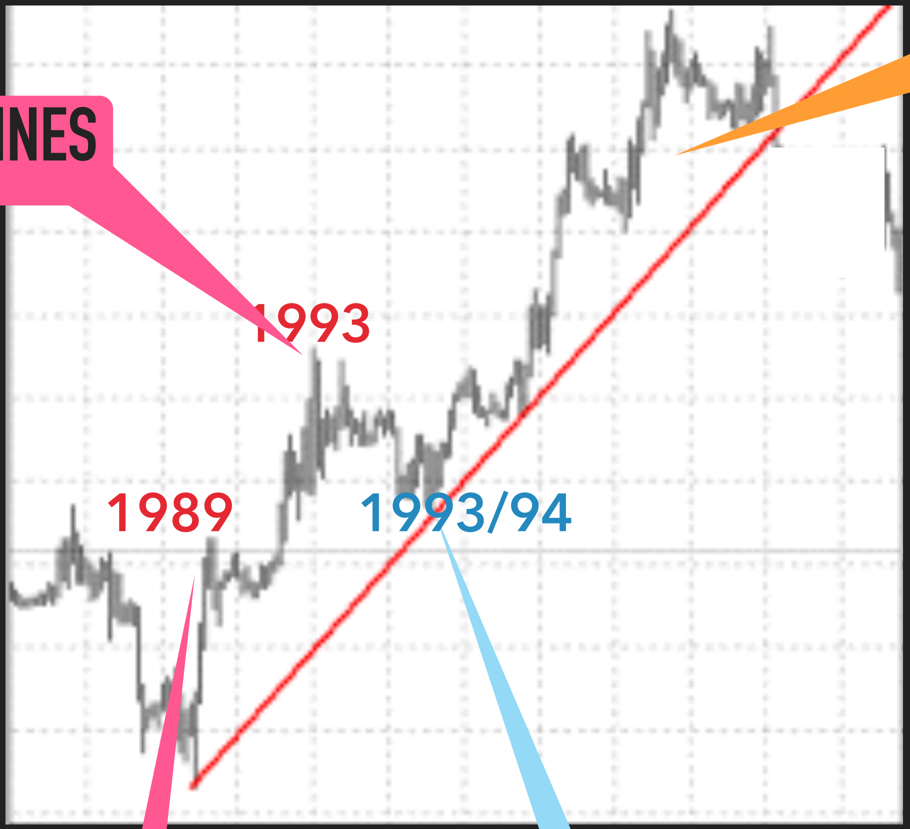
```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (ta
```

The Problem:

Gaussian elimination over equations in an uninterpreted algebras cannot point back to program when the system (of eqs) is inconsistent.

```
any sensible error message,
just one, please .. .. ..
```

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS



WRIGHT CAN CHECK 1,000 LINES

FAGAN CAN CHECK 100 LINES.

YOURS TRULY STRUGGLES WITH DOZENS OF SMALL AND LARGE PROGRAMS.

NEVIN HEINZE SHOWED YOURS TRULY SET-BASED ANALYSIS, AND IT FELT LIKE AN IDEA THAT COULD HELP OUT HERE.

HEYA, DID YOU CATCH THIS MISTAKE?

Catching Bugs in the Web of Program Invariants

Cormac Flanagan   Matthew Flatt   Shriram Krishnamurthi   Stephanie Weirich
Matthias Felleisen

ment of Comp
Rice Univer
ston, Texas 7
ormac@cs.ri

- derive sub-typing constraints from code e.g. dom(f) < rng(g) or int < dom(h)
- solve via the transitive closure *through* the constructors in the constraint algebra
- find type errors by comparing specified constraints for prime with computed ones

Componential Set-Based Analysis

CORMAC FLANAGAN
Compaq Systems Research Center
and
MATTHIAS FELLEISEN
Rice University

WHAT SPIDER CAN DO

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

infer via componential SBA

```
(-> (μ (Proposition)
       (U Boolean (-> Boolean Proposition)))
    Boolean)
```

AND THEY CAN EXPLAIN ERRORS, HALLELUJAH!

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (p false))]))
```

inspect errors via flow graphs and slices drawn on top of code

AND THEY CAN EXPLAIN ERRORS, HALLELUJAH!

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (p false))]))
```

inspect errors via flow graphs and slices drawn on top of code

EVEN WITH 3RD UNDERGRADUATES

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

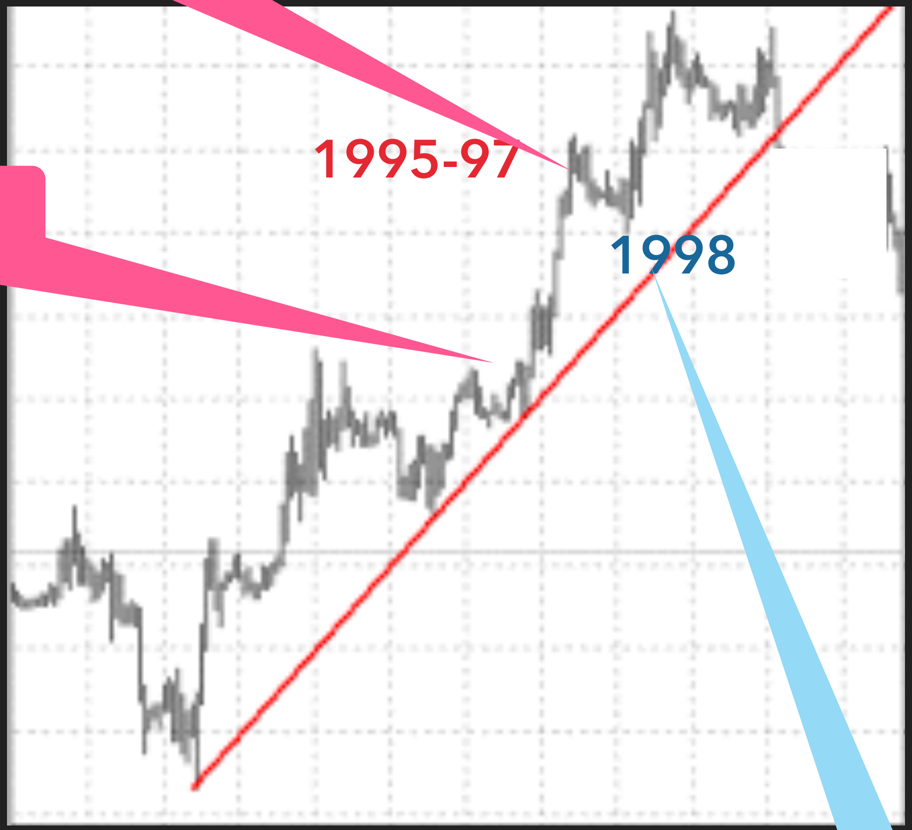
FLANAGAN CAN CHECK 3,000 LINES STUDENT PROGRAMS

FLANAGAN CAN EXPLAIN ERRORS

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

FLANAGAN CAN CHECK 3,000 LINES STUDENT PROGRAMS

FLANAGAN CAN EXPLAIN ERRORS



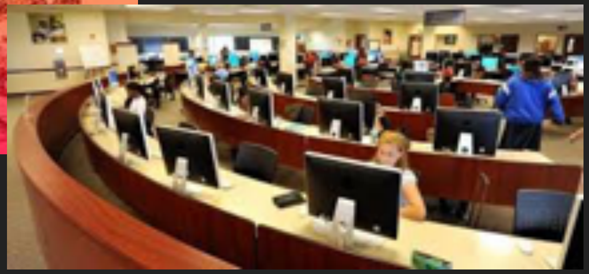
WE CANNOT ANALYZE THE COMPLETE CODE BASE OF THE SYSTEM ITSELF OR ITS CONTEXT. WE CAN'T EVEN 'MODULARIZE' THE ANALYSIS PROPERLY.

WHAT THEY CAN'T DO, ABSOLUTELY NOT, NOT FOR CRYING OUT LOUD

```
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs
```

BUT …

```
an analysis of large
programs or a truly modular
analysis of such systems
```

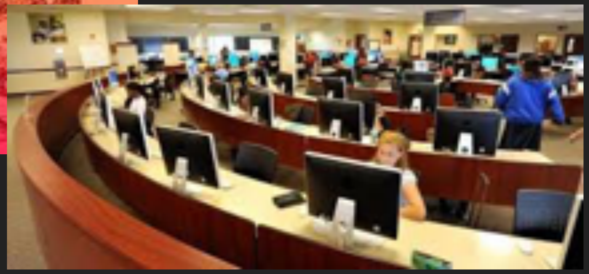


WHAT THEY CAN'T DO, ABSOLUTELY NOT, NOT FOR CRYING OUT LOUD

```
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs
```

BUT ...

an analysis of large programs or a truly modular analysis of such systems

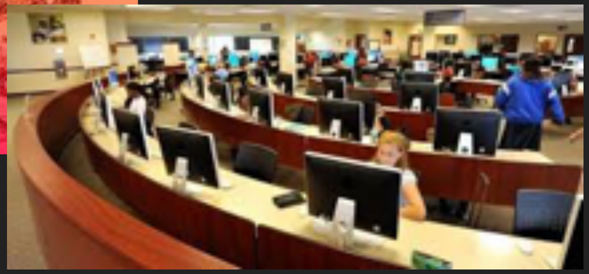


WHAT THEY CAN'T DO, ABSOLUTELY NOT, NOT FOR CRYING OUT LOUD

```
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs
```

WE KNOW TRANSITIVE CLOSURE IS BASICALLY O(N^3) .. BUT ..

BUT ...



an analysis of large programs or a truly modular analysis of such systems

WHAT THEY CAN'T DO, ABSOLUTELY NOT, NOT FOR CRYING OUT LOUD

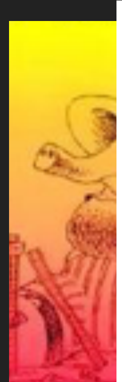
```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

WE KNOW TRANSITIVE CLOSURE IS BASICALLY O(N^3) .. BUT ..

BUT ...

O(n^8)

```
sis of large
 a truly modular
f such systems
```



Selectors Make Set-Based Analysis Too Hard

Philippe Meunier, Robert Bruce Findler†, Paul Steckler and Mitchell Wand

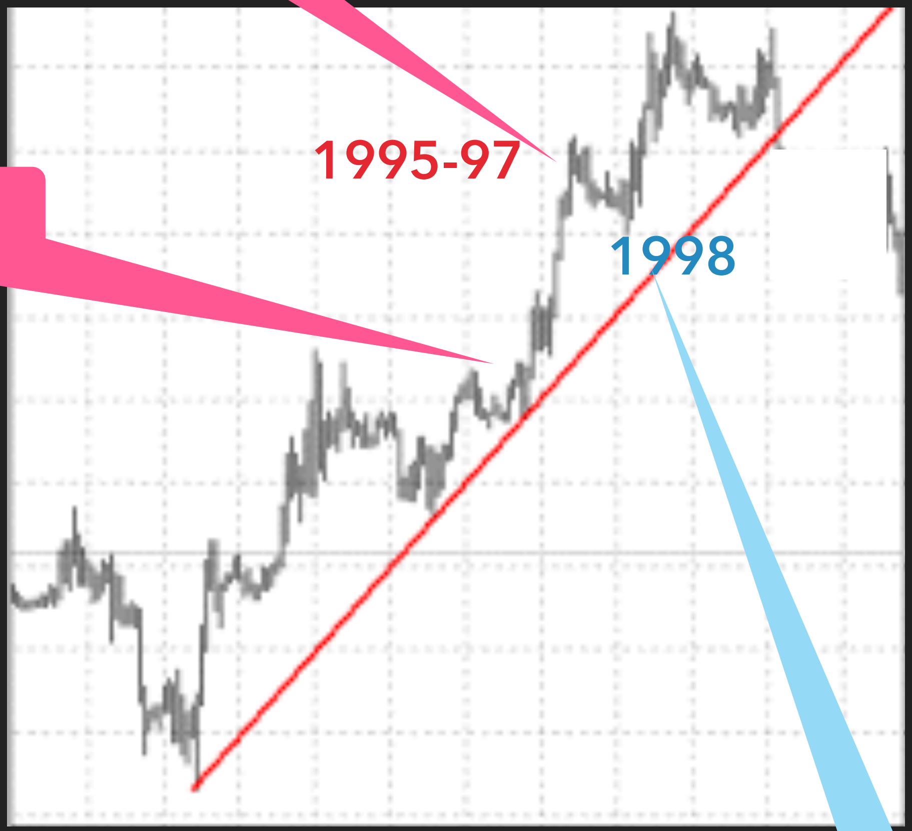
College of Computer and Information Science
Northeastern University
Boston, MA 02115
{meunier,steck,wand}@ccs.neu.edu

† Department of Computer Science
University of Chicago
Chicago, IL 60637
robby@cs.uchicago.edu

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

FLANAGAN CAN CHECK 3,000 LINES

FLANAGAN CAN EXPLAIN ERRORS



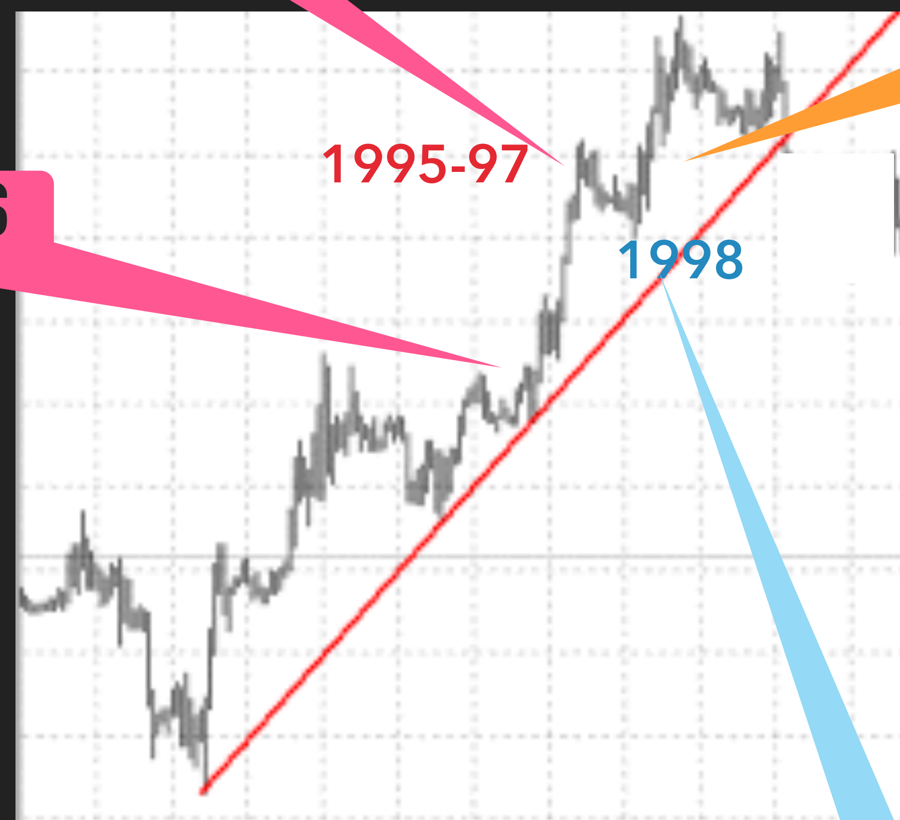
WE CANNOT ANALYZE THE COMPLETE CODE BASE OF THE SYSTEM ITSELF OR ITS CONTEXT. WE CAN'T EVEN 'MODULARIZE' THE ANALYSIS PROPERLY.

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

FLANAGAN CAN CHECK 3,000 LINES

FLANAGAN CAN EXPLAIN ERRORS

YOURS TRULY SAYS PROGRAMMERS USE ASSERTIONS, THEY WILL USE “CONTRACTS.”



WE CANNOT ANALYZE THE COMPLETE CODE BASE OF THE SYSTEM ITSELF OR ITS CONTEXT. WE CAN’T EVEN ‘MODULARIZE’ THE ANALYSIS PROPERLY.

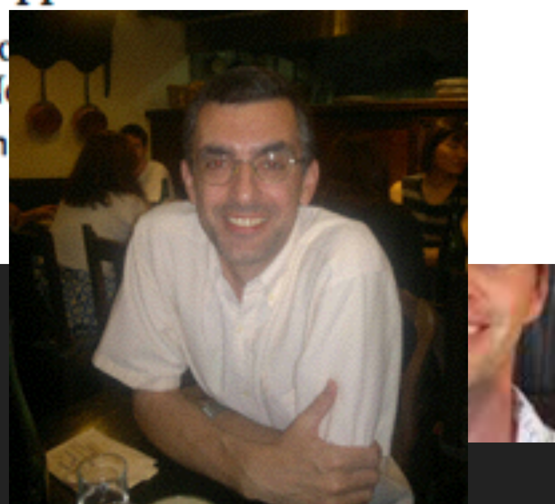
HEYA, DID YOU CATCH THIS MISTAKE?

Modular Set-Based Analysis from Contracts

Philippe Meunier
College of Co
Science, N
meun

Robert Bruce Findler
Department of Computer Science,
University of Chicago
robby@cs.uchicago.edu

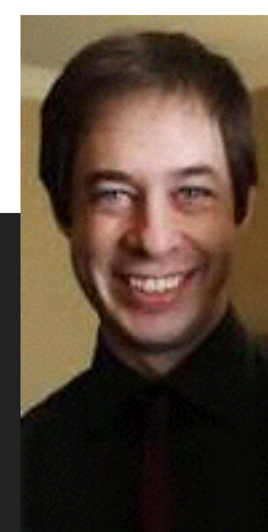
Matthias Felleisen
College of Computer and Information
Science, Northeastern University
matthias@ccs.neu.edu



- modules comes with contracts
- type inference turns contracts into constraints
- .. and stores derived constraints per module

Contracts for Higher-Order Functions

Robert Bruce Findler[1] Matthias Felleisen
Northeastern University
College of Computer Science
ston, Massachusetts 02115, USA

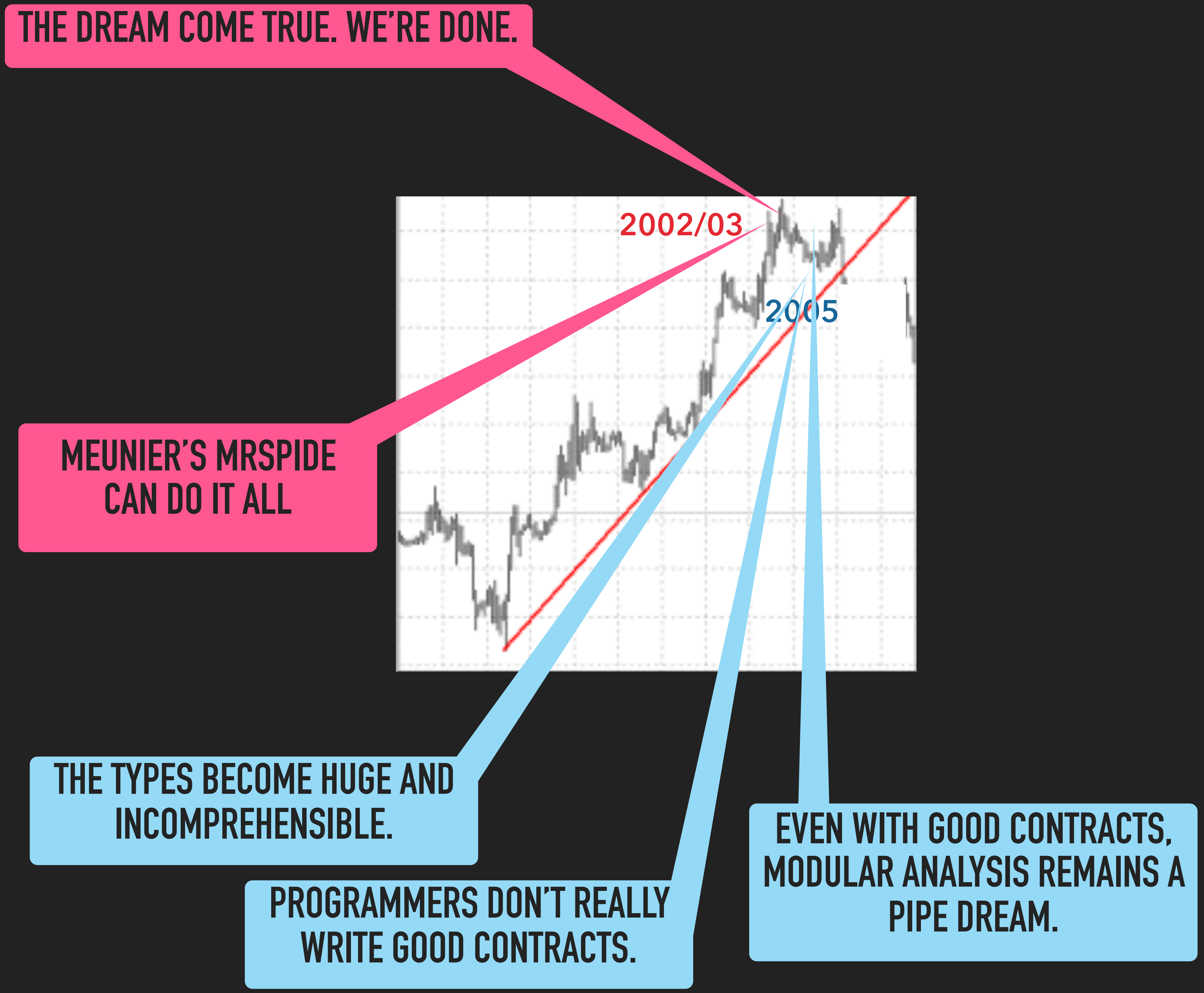


THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE DREAM COME TRUE. WE'RE DONE.

MEUNIER'S MRSPIDE
CAN DO IT ALL

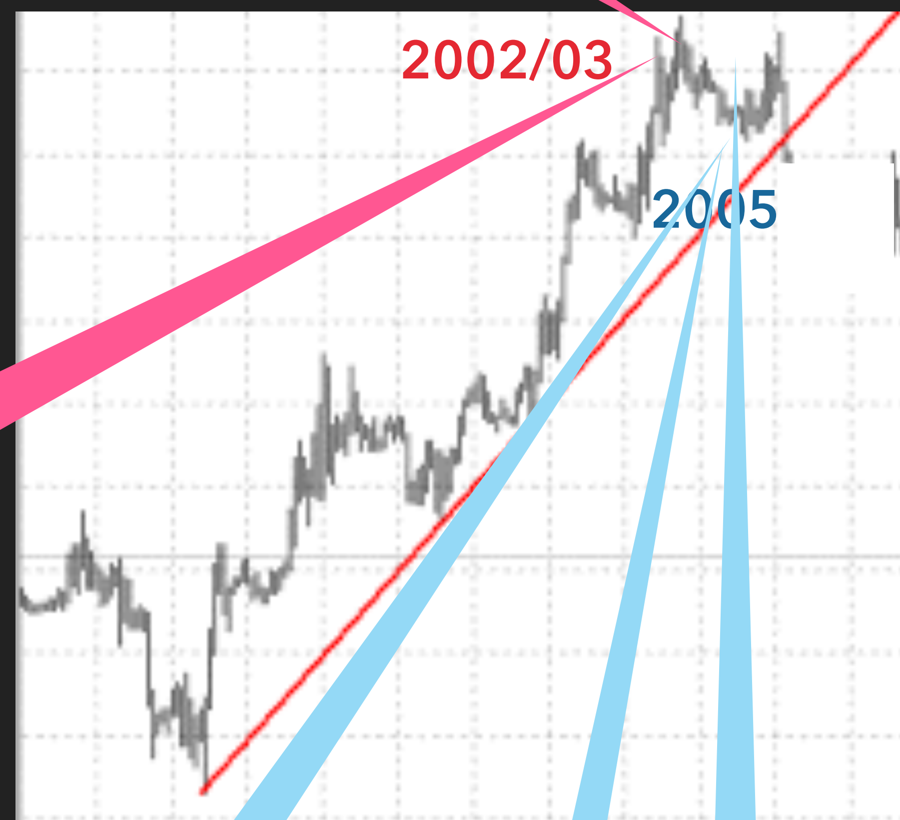
THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS



THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE DREAM COME TRUE. WE'RE DONE.

MEUNIER'S MRSPIDE CAN DO IT ALL



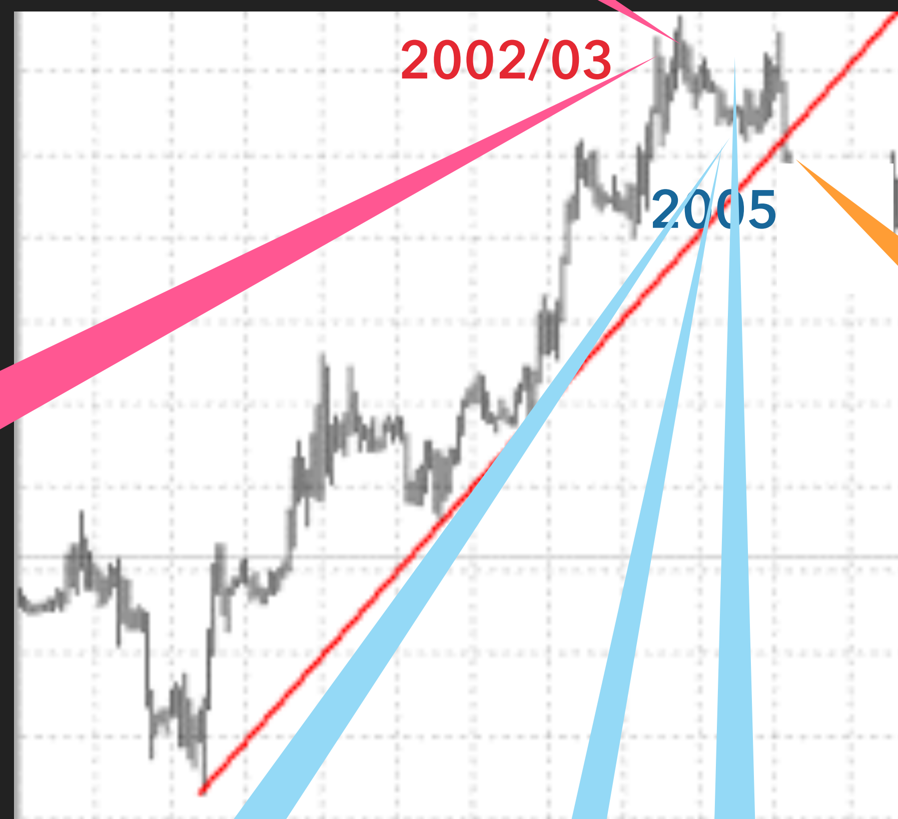
THE TYPES BECOME HUGE AND INCOMPREHENSIBLE.

PROGRAMMERS DON'T REALLY WRITE GOOD CONTRACTS.

EVEN WITH GOOD CONTRACTS, MODULAR ANALYSIS REMAINS A PIPE DREAM.

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE DREAM COME TRUE. WE'RE DONE.



MEUNIER'S MRSPIDE CAN DO IT ALL

YOURS TRULY SAYS "NEVER MIND."

THE TYPES BECOME HUGE AND INCOMPREHENSIBLE.

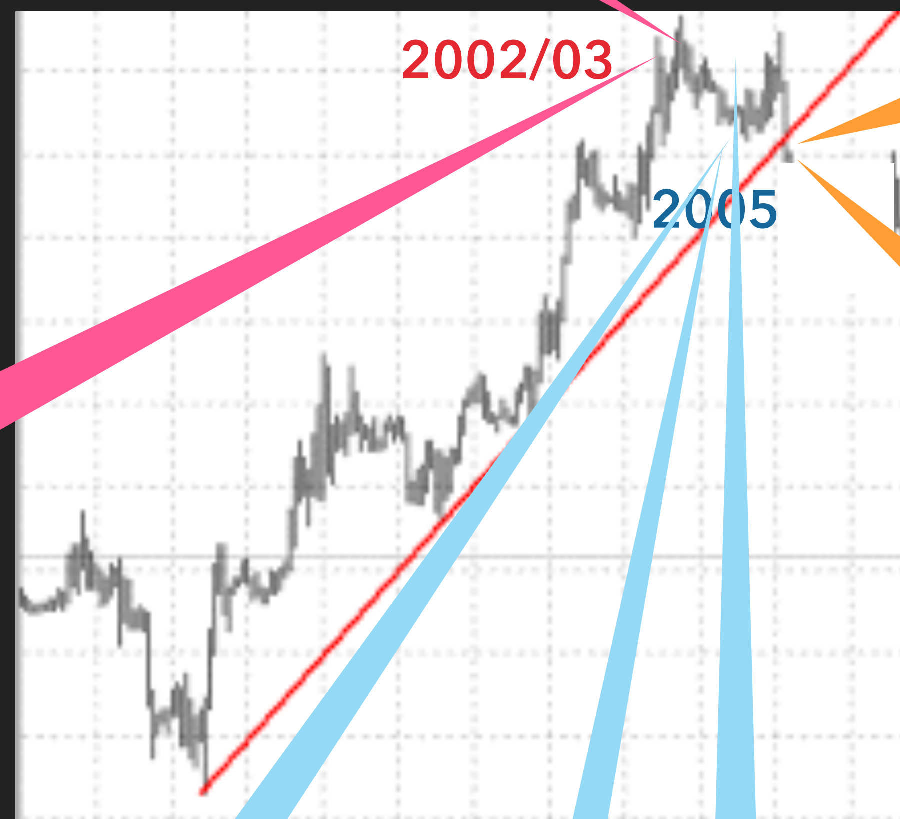
PROGRAMMERS DON'T REALLY WRITE GOOD CONTRACTS.

EVEN WITH GOOD CONTRACTS, MODULAR ANALYSIS REMAINS A PIPE DREAM.

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE DREAM COME TRUE. WE'RE DONE.

THE PROBLEM HAD BECOME REAL. IT WAS TIME TO THINK DIFFERENTLY



MEUNIER'S MRSPIDE CAN DO IT ALL

YOURS TRULY SAYS "NEVER MIND."

THE TYPES BECOME HUGE AND INCOMPREHENSIBLE.

PROGRAMMERS DON'T REALLY WRITE GOOD CONTRACTS.

EVEN WITH GOOD CONTRACTS, MODULAR ANALYSIS REMAINS A PIPE DREAM.

LET'S ADD TYPES INCREMENTALLY
TO A CODE BASE AND MAKE SURE
THE COMBINATION IS SOUND.

ADD TYPES INCREMENTALLY

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true))
               (tautology? (p false)))]))
```

```
lambda (x)
        (or x y))))
   false)
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology?
(p true))
               (tautology?
(p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true))
               (tautology? (p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

ADD TYPES INCREMENTALLY

- You want to add types.
- And now you have two problems:
  - You should not change code that works, other than adding type annotations and definitions. *Respect existing idioms of the language.*
  - You want the existing untyped code to play well with the newly typed code. *Respect the central theorem of programming languages, type soundness.*

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

        (or x y))))

   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)
```

```
lambda (x)

        (or x y))))

   false)

(define (tautology? p)

 p]

autology? (p true))

autology? (p false)))]))
```

```
lambda (x)

        (or x y))))

   false)
```

```
lean -> Proposition)

da (_) true)) true)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
     [else (and (tautology?
(p true))

                (tautology?
(p false)))]))
```

```
(check-expect

   (tautology?
```

```
sition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

ADD TYPES INCREMENTALLY

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

       (or x y))))

   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

       (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))

               (tautology? (p false)))]))
```

```
lambda (x)

       (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology?
(p true))

               (tautology?
(p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

       (or x y))))

   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

       (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))

               (tautology? (p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

ADD TYPES INCREMENTALLY

```
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

ADD TYPES INCREMENTALLY

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean
```

```
(define-type Proposition (U Boolean (Boolean -> Proposition)))


(: tautology? (-> Proposition Boolean))
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)
(tautology? (lambda ({x : Boolean}) (lambda ({y : Boolean}) (or x y))))
```

ADD TYPES INCREMENTALLY

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true))
               (tautology? (p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
   false)
```

```
lambda (x)
        (or x y))))
   false)
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true))
               (tautology? (p false)))]))
```

```
lambda (x)
        (or x y))))
   false)
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? 
(p true))
               (tautology? 
(p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

ADD TYPES INCREMENTALLY

Interlanguage Migration: From Scripts to Programs

Sam Tobin-Hochstadt
Northeastern University
Boston, MA
samth@ccs.neu.edu

Matthias Felleisen
Northeastern University
Boston, MA
matthias@ccs.neu.edu

The Design and Implementation of Typed Scheme

Sam Tobin-Hochstadt Matthias Felleisen
PLT, Northeastern University
Boston, MA 02115

Logical Types for Untyped Languages *

Sam Tobin-Hochstadt Matthias Felleisen
Northeastern University
{samth,matthias}@ccs.neu.edu

ADD TYPES INCREMENTALLY: ACCOMMODATING GROWN PROGRAMMING IDIOMS

```
(define-type Proposition (U Boolean (Boolean -> Proposition)))
(: tautology? (Proposition -> Boolean))
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
       (or x y))))
   false)
(define (tautology? p)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
```

```
lambda (x)
       (or x y))))
   false)
(define (tautology? p)
  (cond
```

```
lambda (x
false)
```

```
;; Propos
;; Propos
(check-ex
(check-ex
   (tautology?
```

```
    [(boolean? p) p]
    [else (and (tautology?
(p true))
                (tautology?
(p false)))]))
```

```
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

ADD TYPES INCREMENTALLY: ACCOMMODATING GROWN PROGRAMMING IDIOMS

```
(define-type Proposition (U Boolean (Boolean -> Proposition)))

(: tautology? (Proposition -> Boolean))

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

: PROPOSITION

ADD TYPES INCREMENTALLY: ACCOMMODATING GROWN PROGRAMMING IDIOMS

```
(define-type Proposition (U Boolean (Boolean -> Proposition)))
(: tautology? (Proposition -> Boolean))
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

: PROPOSITION

: BOOLEAN

ADD TYPES INCREMENTALLY: ACCOMMODATING GROWN PROGRAMMING IDIOMS

```
(define-type Proposition (U Boolean (Boolean -> Proposition)))
(: tautology? (Proposition -> Boolean))
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

: PROPOSITION

: BOOLEAN

: (-> BOOLEAN PROPOPSITION)

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
lambda (x)
        (or x y))))
    false)
(define (tautology? p)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
```

```
lambda (x)
        (or x y))))
    false)
(define (tautology? p)
  (cond
```

```
    [(boolean? p) p]
    [else (and (tautology?
(p true))
               (tautology?
(p false)))]))
```

```
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

```
;; Proposition = Boolean | (Boolean -> Proposition)
;; Proposition -> Boolean
(check-expect (tautology? (lambda (_) true)) true)
(check-expect
   (tautology?
```

ADD TYPES INCREMENTALLY: ACCOMMODATING GROWN PROGRAMMING IDIOMS

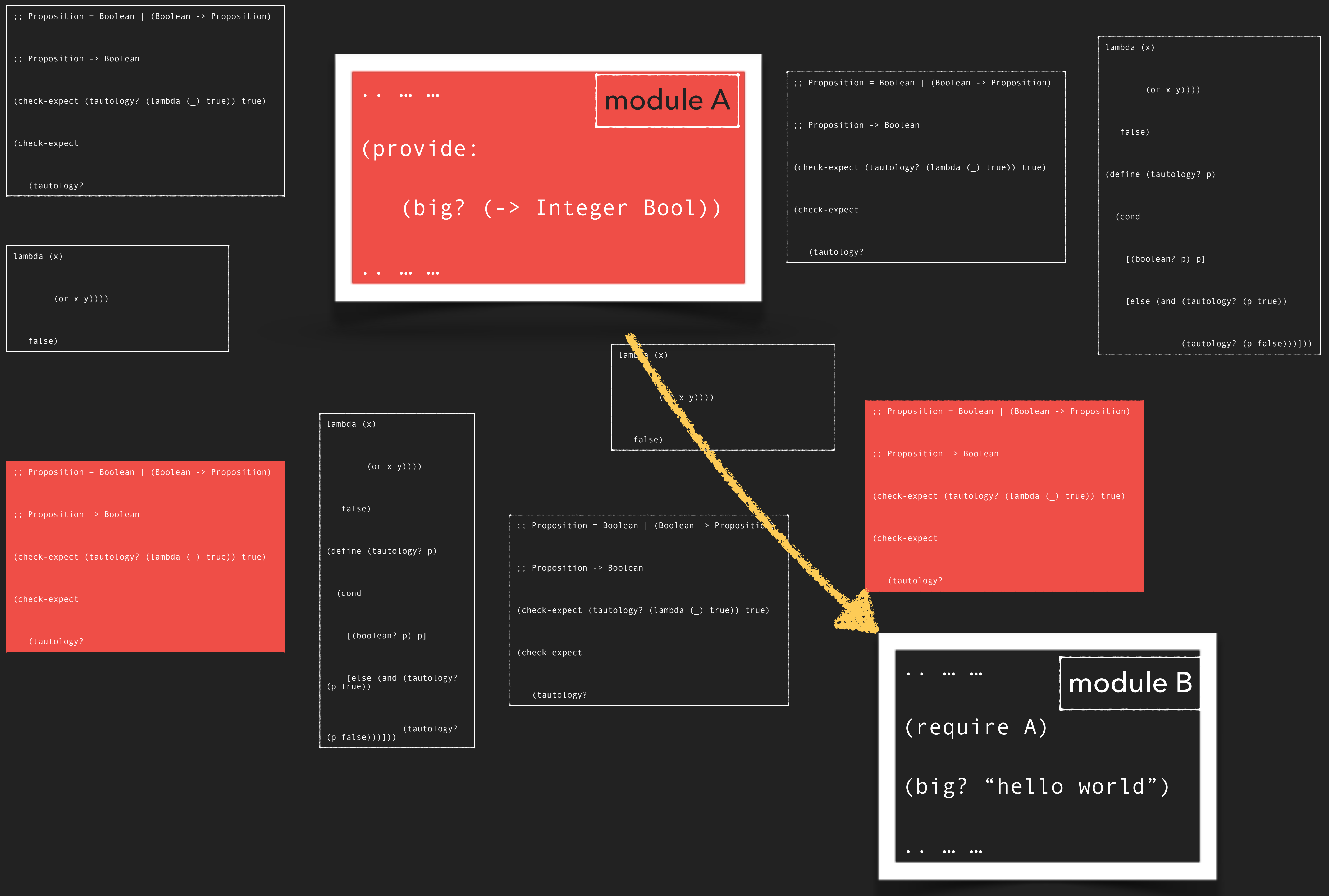
```
(define-type Proposition (U Boolean (Boolean -> Proposition)))
(: tautology?
(define (tau
  (cond
    [(boolea
    [else (and (tautology? (p true)) (tautology? (p false)))]))
```

Logical Types for Untyped Languages *

Sam Tobin-Hochstadt Matthias Felleisen
Northeastern University
{samth,matthias}@ccs.neu.edu

ICFP 2010

ADD TYPES INCREMENTALLY: PROTECTION FROM CONTRACTS



ADD TYPES INCREMENTALLY: PROTECTION FROM CONTRACTS

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

module A

```
. .  ...  ...

(provide:

    (big? (-> Integer Bool))

. .  ...  ...
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

        (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))

                    (tautology? (p false)))]))
```

```
lambda (x)

        (or x y))))

   false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

WHAT PREVENTS MODULE B FROM APPLYING THE BIG? FUNCTION TO A STRING?

CONTRACTS!

```
  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))

                    (tautology? (p false)))]))
```

```
(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

module B

```
. .  ...  ...

(require A)

(big? "hello world")

. .  ...  ...
```

ADD TYPES INCREMENTALLY: PROTECTION FROM CONTRACTS

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

module A

```
. .  …  …

(provide:

   (big? (-> Integer Bool))

. .  …  …
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

      (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))

                    (tautology? (p false)))]))
```

```
lambda (x)

      (or x y))))

   false)
```

Interlanguage Migration: From Scripts to Programs

Sam Tobin-Hochstadt
Northeastern University
Boston, MA
samth@ccs.neu.edu

DLS 2006

Matthias Felleisen
Northeastern University
Boston, MA
matthias@ccs.neu.edu

```
;; Proposition =

;; Proposition ->

(check-expect (ta

(check-expect

   (tautology?
```

```
    [(boolean? p) p]

    [else (and (tautology?
(p true))

               (tautology?
(p false)))]))
```

```
(check-expect

   (tautology?
```

module B

```
. .  …  …

(require A)

(big? "hello world")

. .  …  …
```

ADD TYPES INCREMENTALLY: PROTECTION FROM CONTRACTS

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

        (or x y))))

   false)
```

```
. .  ...  ...                          module A

(provide:

    (big? (-> Integer Bool))

. .  ...  ...
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

        (or x y))))

     false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))

               (tautology? (p false)))]))
```

```
lambda (x)

         (or x y))))

     false)
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

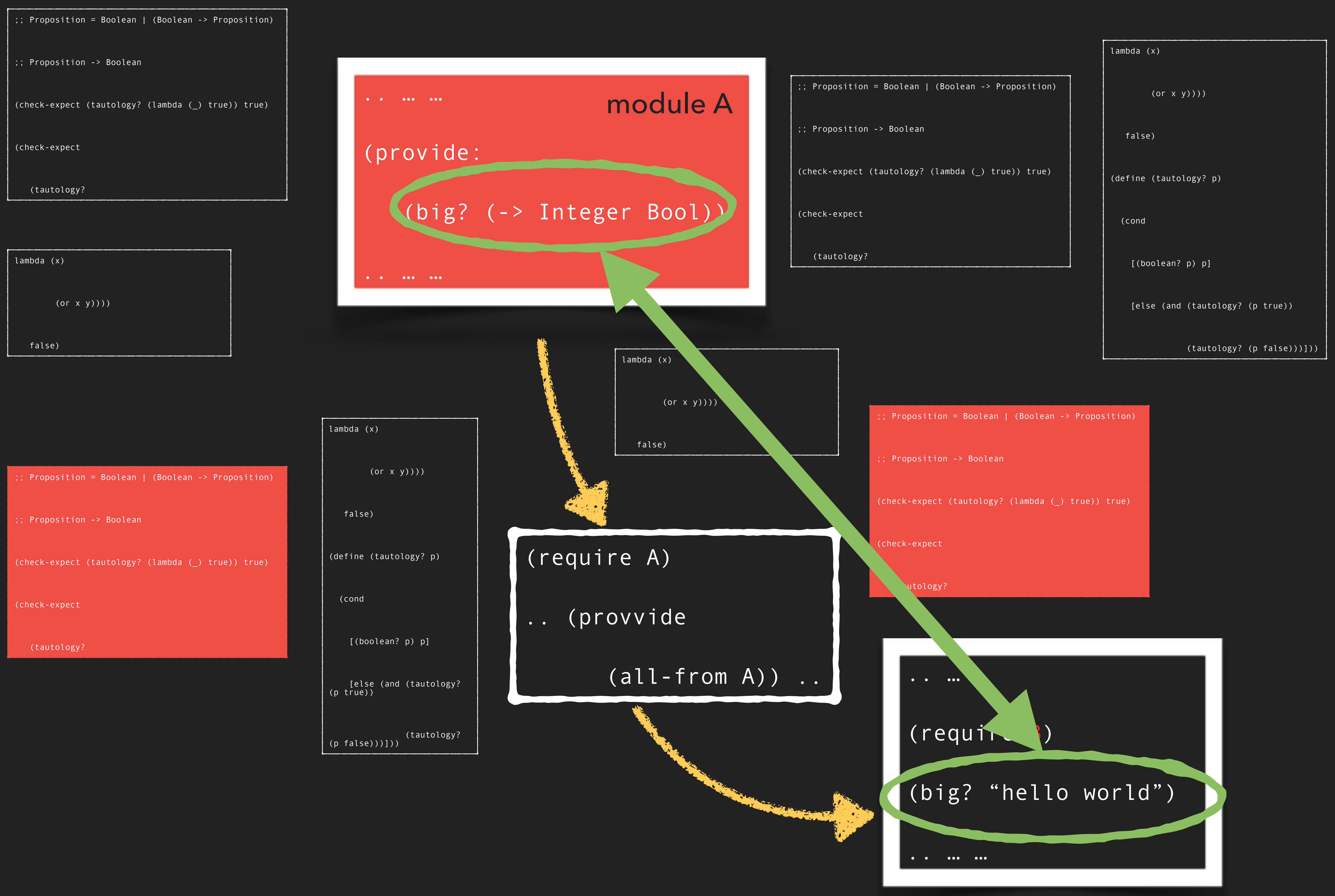
```
lambda (x)

        (or x y))))

     false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology?
(p true))

               (tautology?
(p false)))]))
```

```
(require A)

.. (provvide

         (all-from A)) ..
```

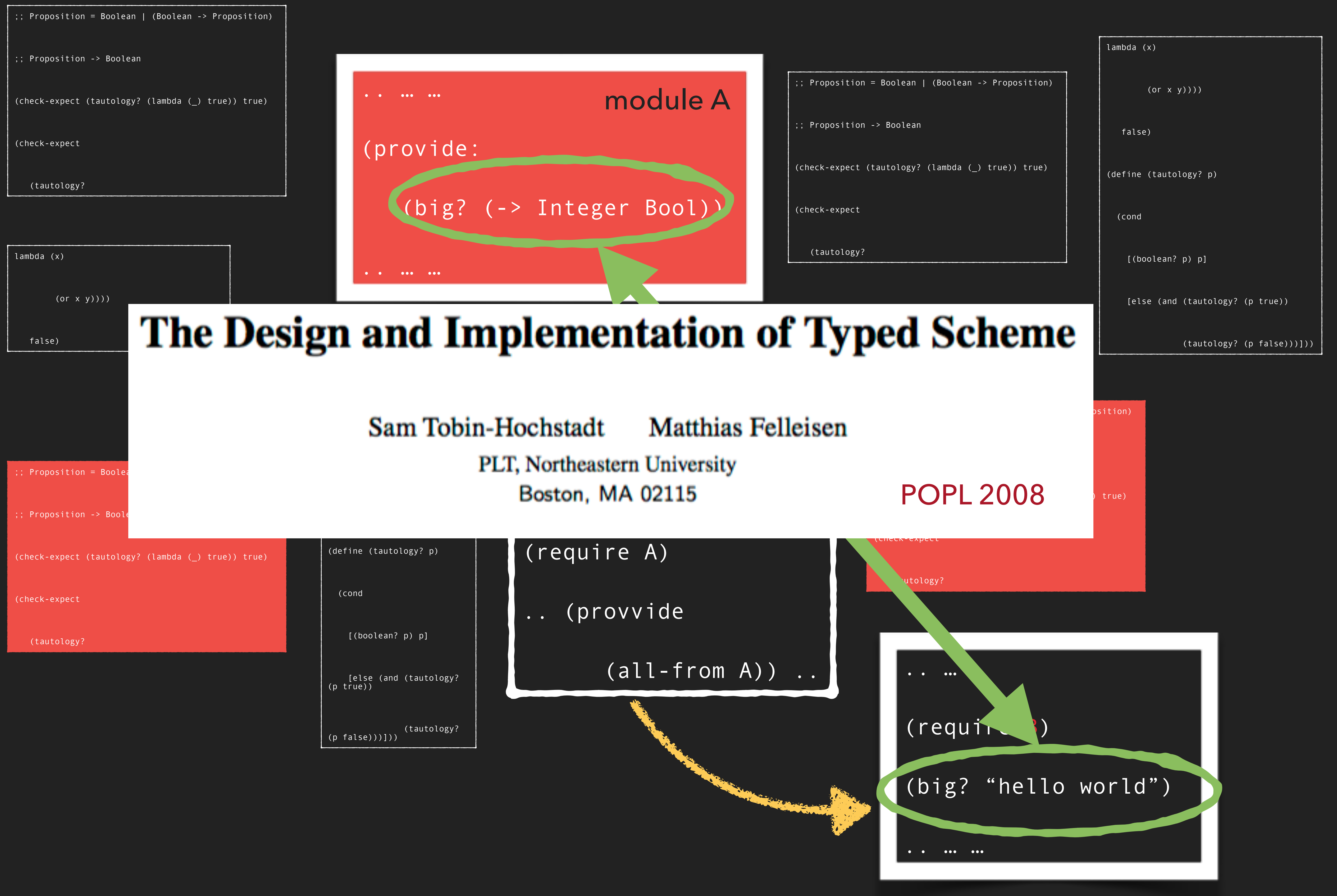
```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
. .  ...  ...

(require B)

(big? “hello world”)

. .  ...  ...
```

ADD TYPES INCREMENTALLY: PROTECTION FROM CONTRACTS



ADD TYPES INCREMENTALLY: PROTECTION FROM CONTRACTS



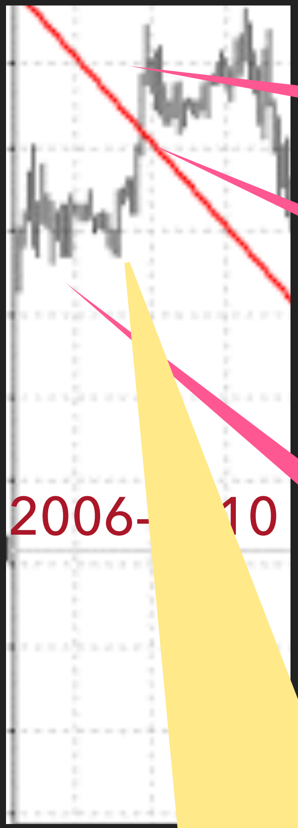
THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

THE IDEA IS WORKED OUT

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS



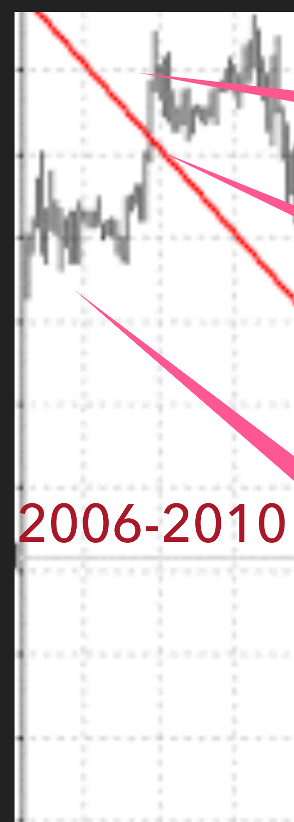
THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

THE IDEA IS WORKED OUT

WHAT'S MISSING?

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

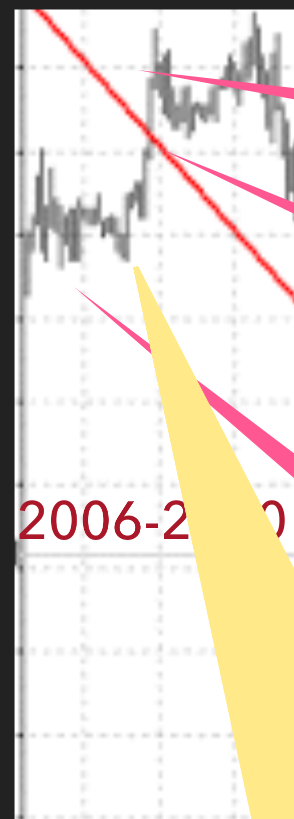


THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

THE IDEA IS WORKED OUT

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS



THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

THE IDEA IS WORKED OUT

OBJECT-ORIENTED RACKET

THERE IT IS!

Gradual Typing for First-Class Classes *

Asumu Takikawa   T. Stephen Strickland   Christos Dimoulas
Sam Tobin-Hochstadt   Matthias Felleisen
PLT, Northeastern University
{asumu, sstrickl, chrdimo, samth, matthias}@ccs.neu.edu

THE EXISTING IDIOMS.

TRACTS WORK

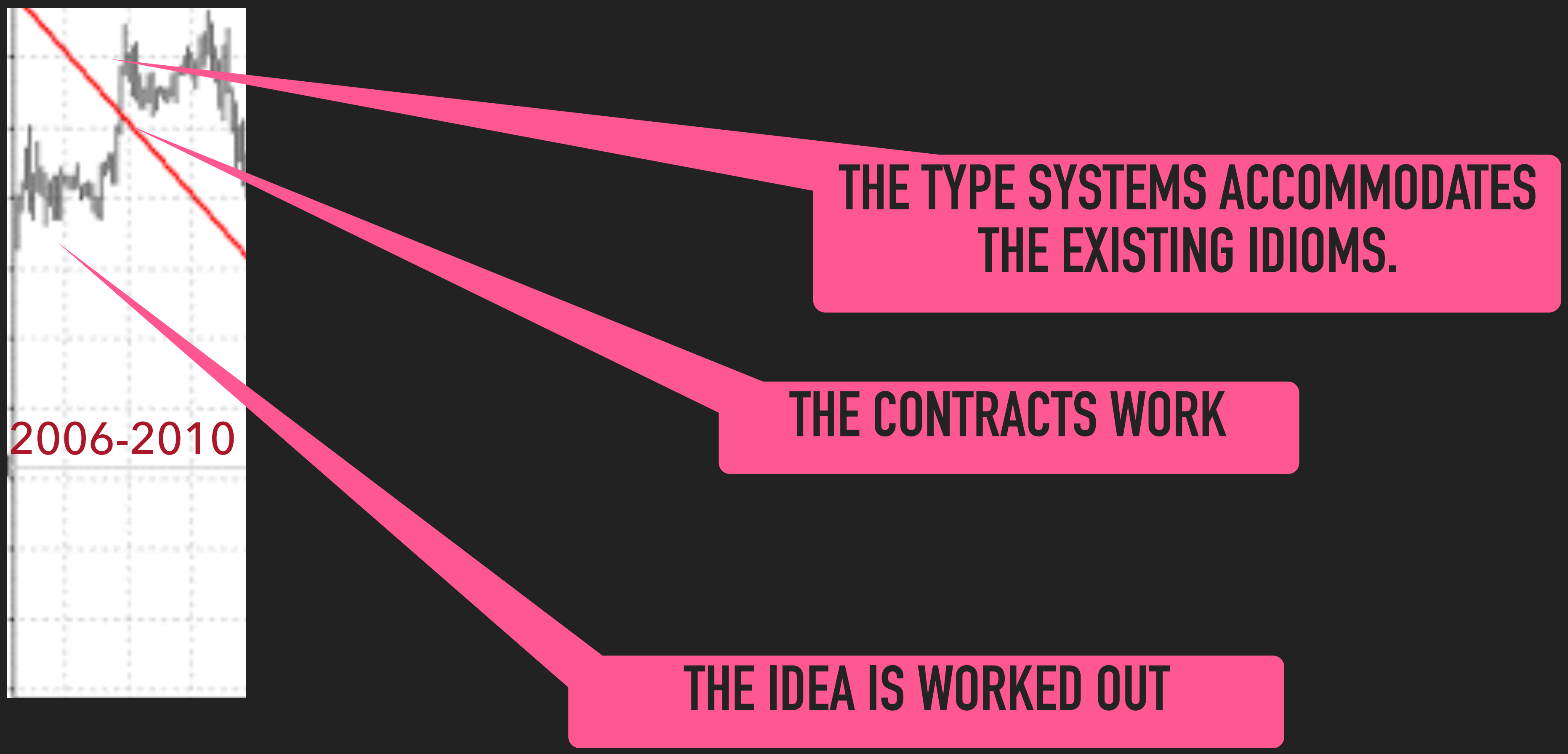
ORKED OUT

Towards Practical Gradual Typing*

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean[2], Matthew Flatt[3], Robert Bruce Findler[4], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

1 Northeastern University
Boston, Massachusetts
asumu@ccs.neu.edu, dfeltey@ccs.neu.edu, matthias@ccs.neu.edu
2 Indiana University
Bloomington, Indiana
samth@cs.indiana.edu, edean@cs.indiana.edu
3 University of Utah
Salt Lake City, Utah
mflatt@cs.utah.edu
4 Northwestern University
Evanston, Illinois
robby@eecs.northwestern.edu

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS



THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

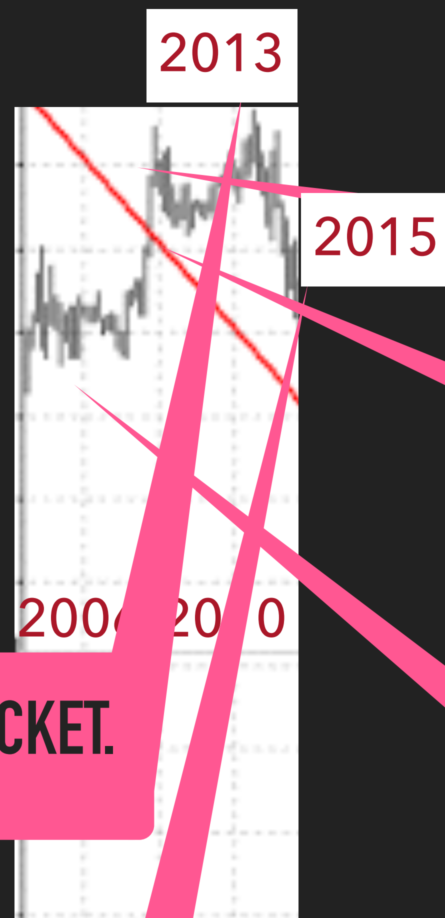
THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

WE HAVE A DESIGN FOR OO RACKET.

THE IDEA IS WORKED OUT

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS



THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

THE IDEA IS WORKED OUT

WE HAVE A DESIGN FOR OO RACKET.

WE HAVE AN IMPLEMENTATION.

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

EVERYTHING HUNKY DORY?

THE TYPE SYSTEMS ACCOMMODATES THE EXISTING IDIOMS.

THE CONTRACTS WORK

THE IDEA IS WORKED OUT

WE HAVE A DESIGN FOR OO RACKET.

WE HAVE AN IMPLEMENTATION.

ADDING TYPES INCREMENTALLY: PERFORMANCE MEASUREMENTS

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

        (or x y))))
```

```
lambda (x)

        (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))
```

```
;; Proposition = Boolean | (Boolean -> Proposition)

;; Proposition -> Boolean

(check-expect (tautology? (lambda (_) true)) true)

(check-expect

   (tautology?
```

```
lambda (x)

        (or x y))))

   false)

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true))
```

```
;;

;;

(ch

(ch
```

Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen
Northeastern University, Boston, MA

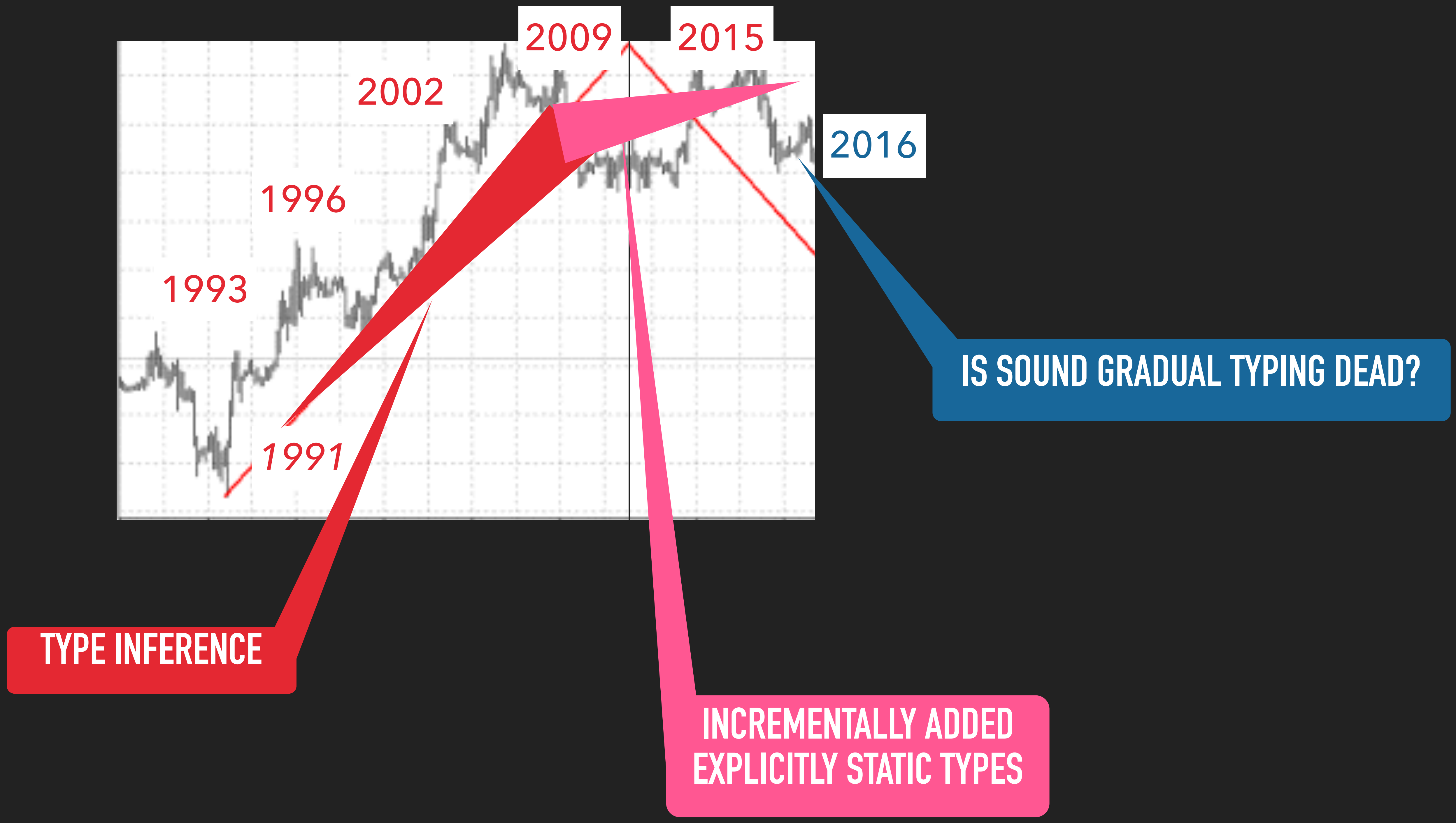
POPL 2016

THE UPS AND DOWNS OF ONE OF MY OWN RESEARCH TOPICS

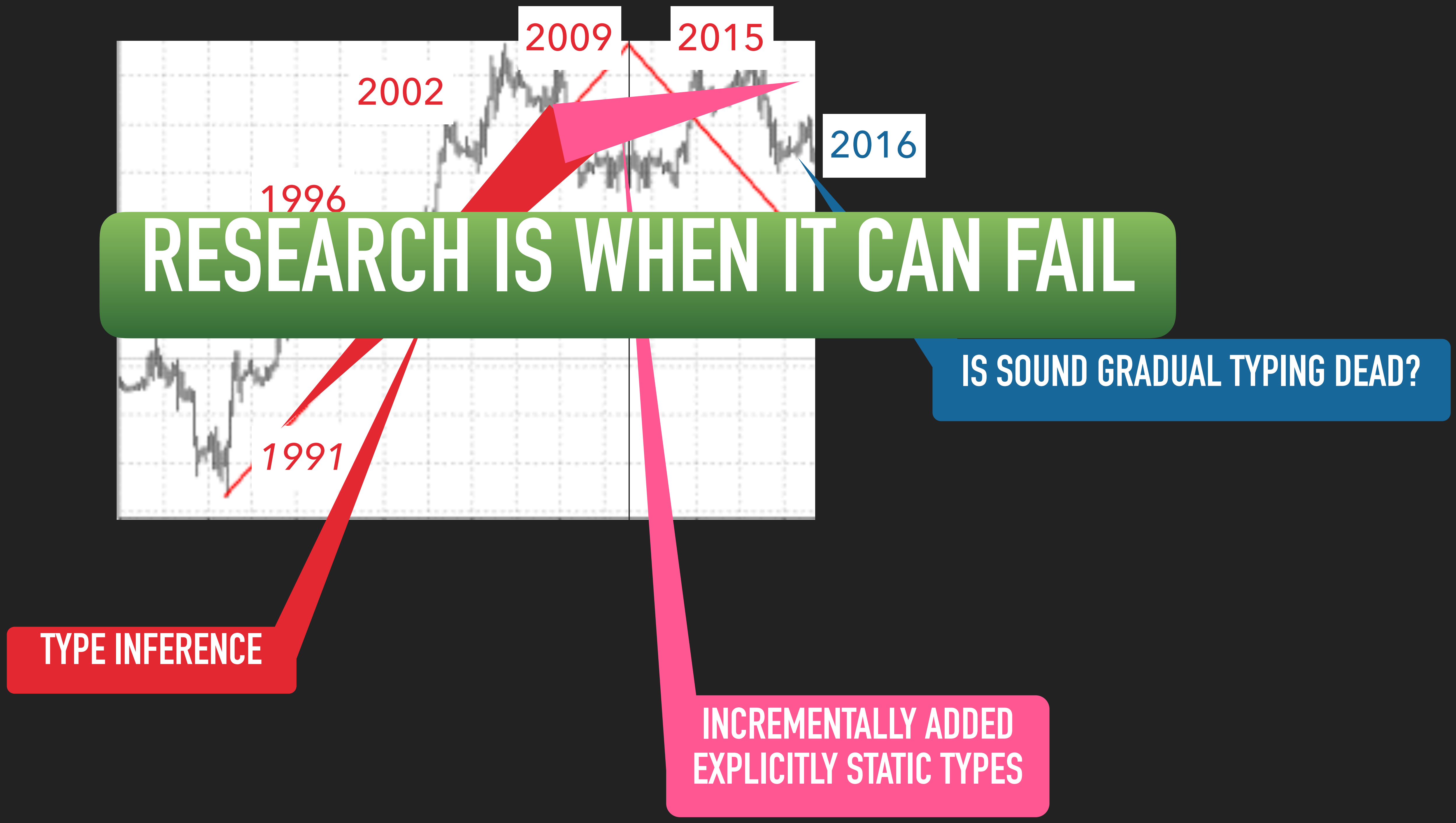
PERFORMANCE. PERFORMANCE. PERFORMANCE.

BACK TO THIS LOVE AND PHD BUSINESS.

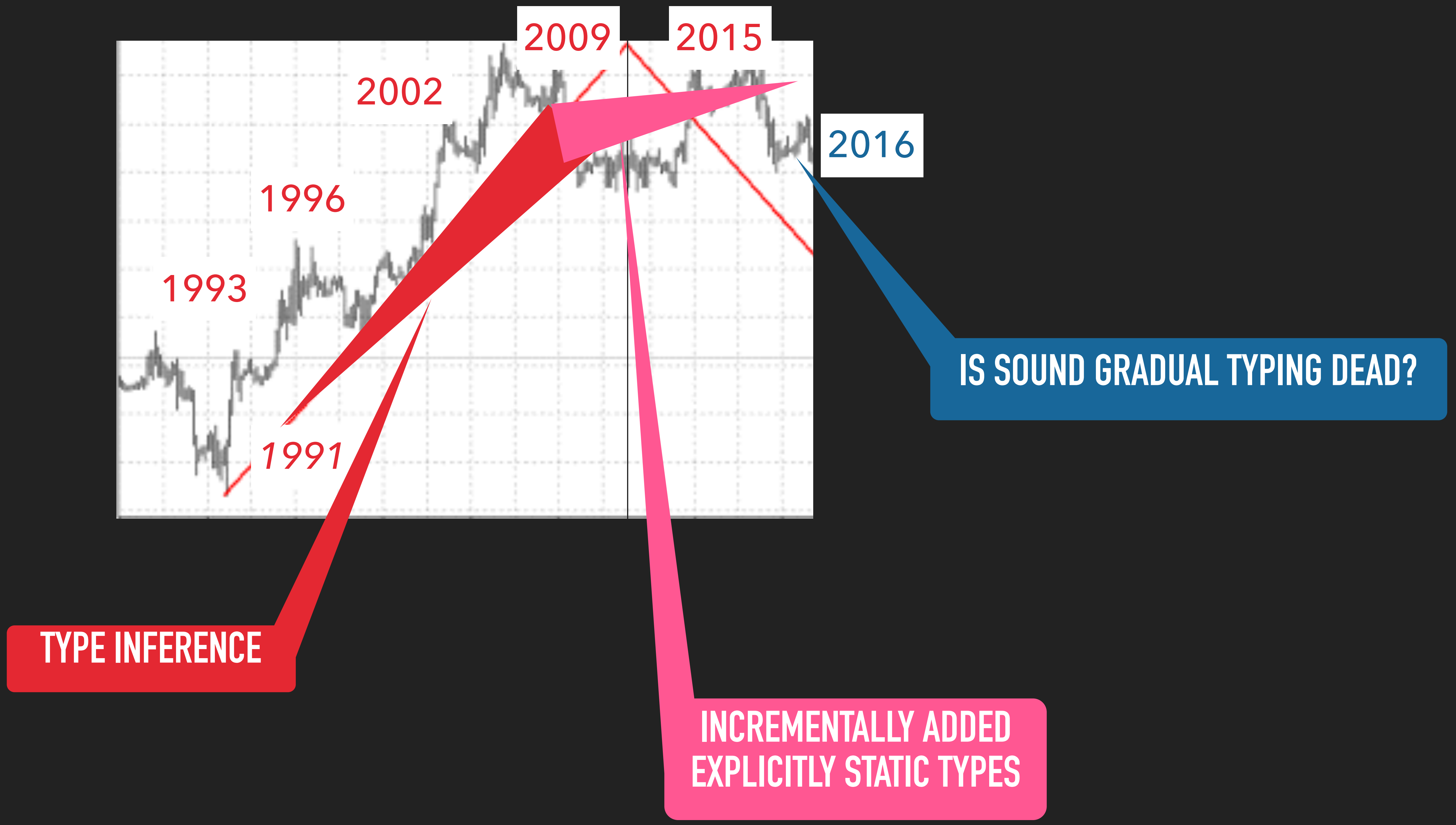
WHAT DO YOU DO WHEN YOU GET INTO SUCH A BAD SITUATION?



WHAT DO YOU DO WHEN YOU GET INTO SUCH A BAD SITUATION?



WHAT DO YOU DO WHEN YOU GET INTO SUCH A BAD SITUATION?



WHAT DO YOU DO WHEN YOU GET INTO SUCH A BAD SITUATION?

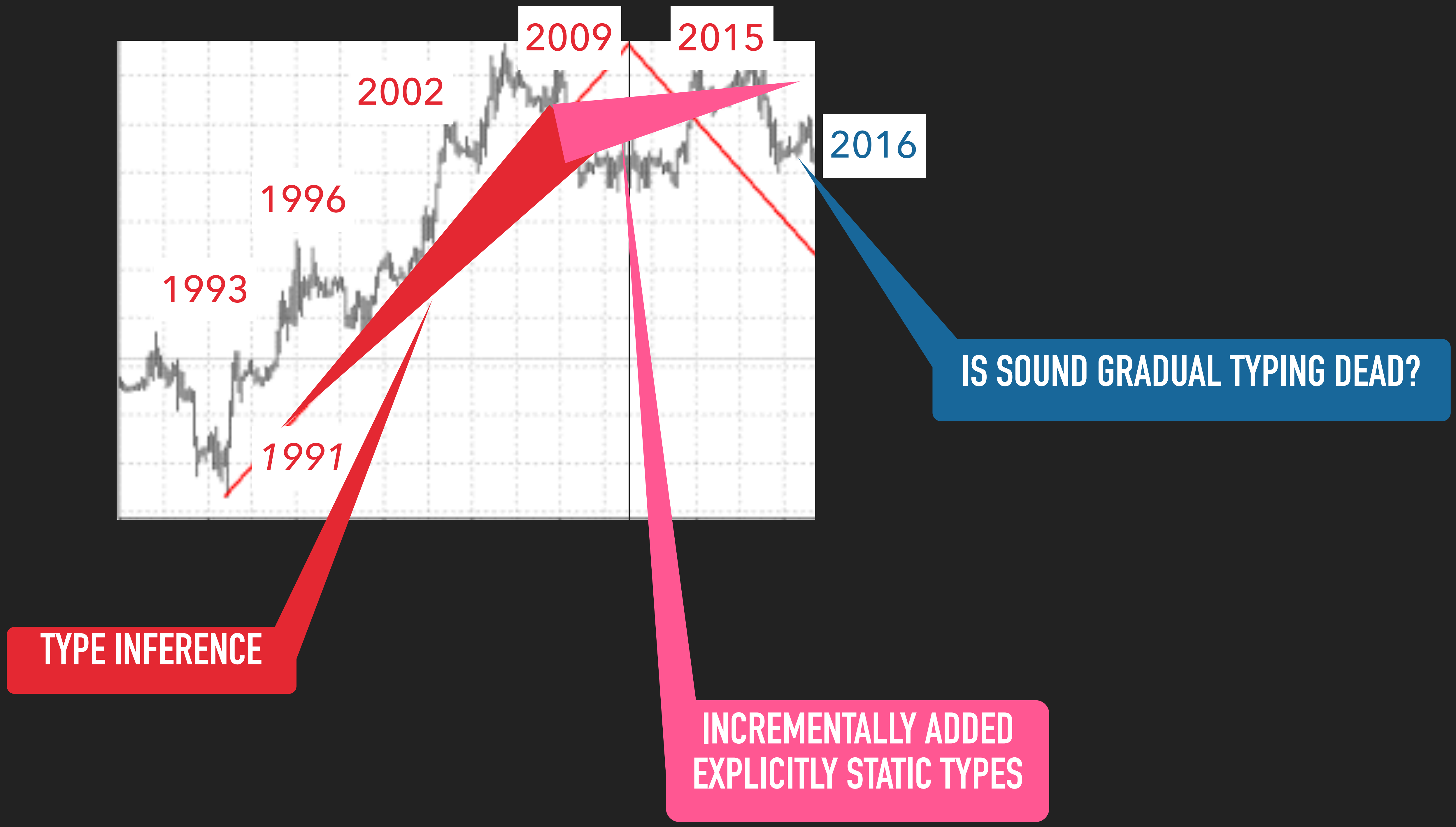
THIS HAPPENS TO ADVISORS WITH A LONG-TERM RESEARCH PROGRAM

UAL TYPING DEAD?

TYPE INFERENCE

INCREMENTALLY ADDED EXPLICITLY STATIC TYPES

WHAT DO YOU DO WHEN YOU GET INTO SUCH A BAD SITUATION?



WHAT DO YOU DO WHEN YOU GET INTO SUCH A BAD SITUATION?

AND IT HAPPENS TO STUDENTS DURING A PHD PROGRAM.

UAL TYPING DEAD?

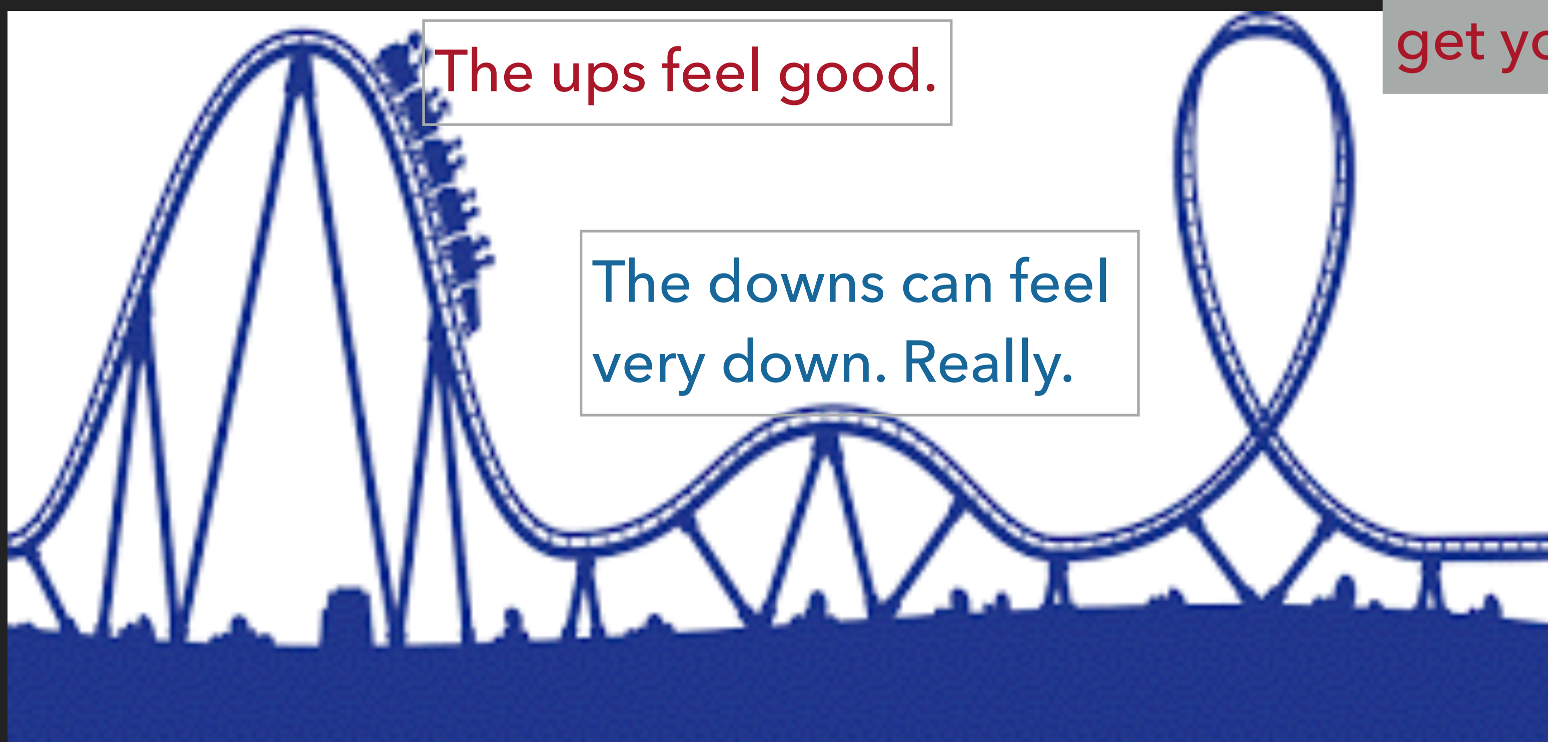
TYPE INFERENCE

INCREMENTALLY ADDED EXPLICITLY STATIC TYPES

PHD RESEARCH: BE PREPARED FOR UPS AND DOWNS

PhD research, like a relationship, has its ups and downs.

The memories of “falling in love” can get you going again.



The ups feel good.

The downs can feel very down. Really.

PHD RESEARCH: BE PREPARED FOR UPS AND DOWNS

PhD research, like a relationship, has its ups and downs.

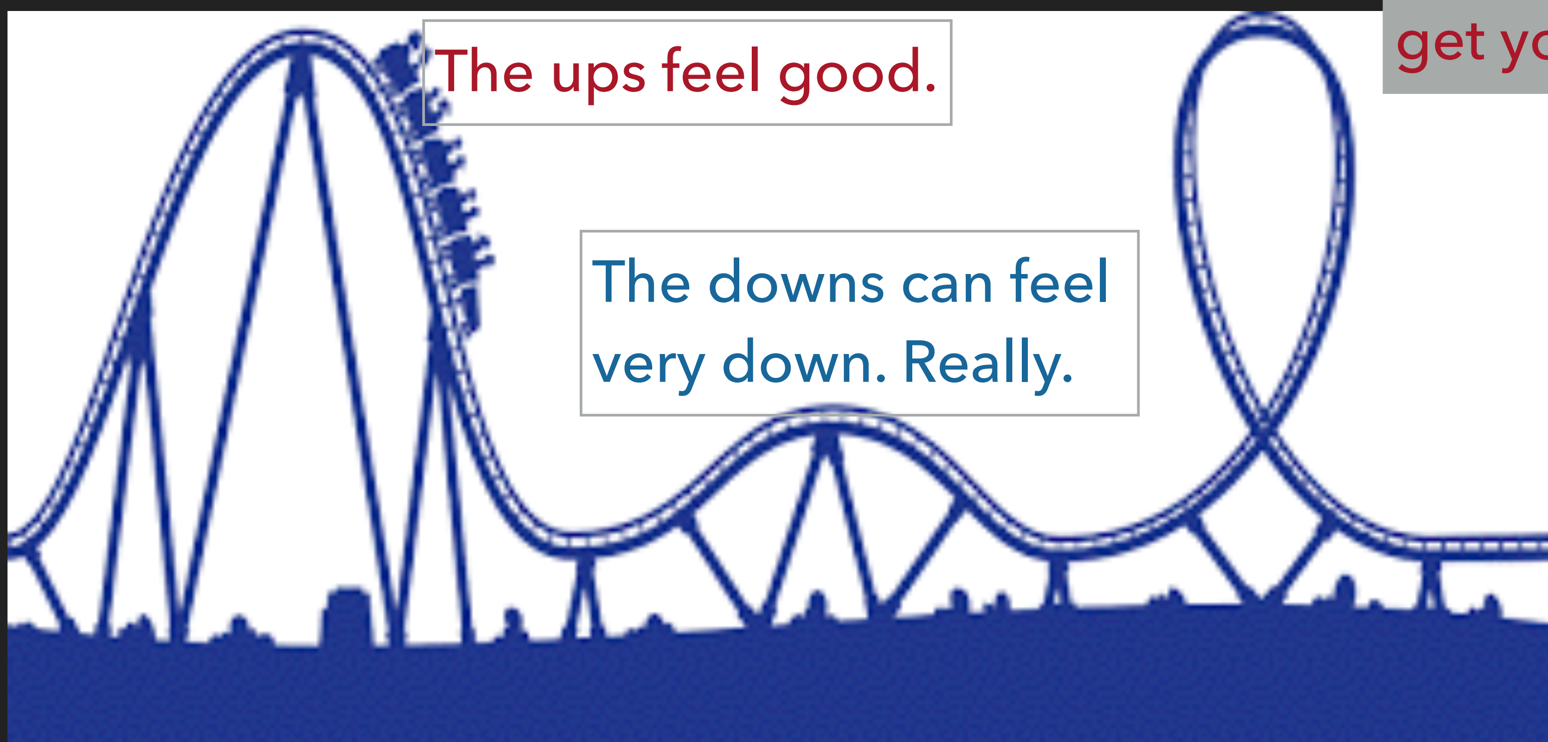
The memories of "falling in love" can get you going again.

The ups feel good.

YES, THIS IS IRRATIONAL BUT WHILE DESIGN, ENGINEERING, & SCIENCE PRODUCE RATIONAL RESULTS, THE MOTIVATION NEEDS AN IRRATIONAL ELEMENT.

PHD RESEARCH: BE PREPARED FOR UPS AND DOWNS

PhD research, like a relationship, has its ups and downs.



The ups feel good.

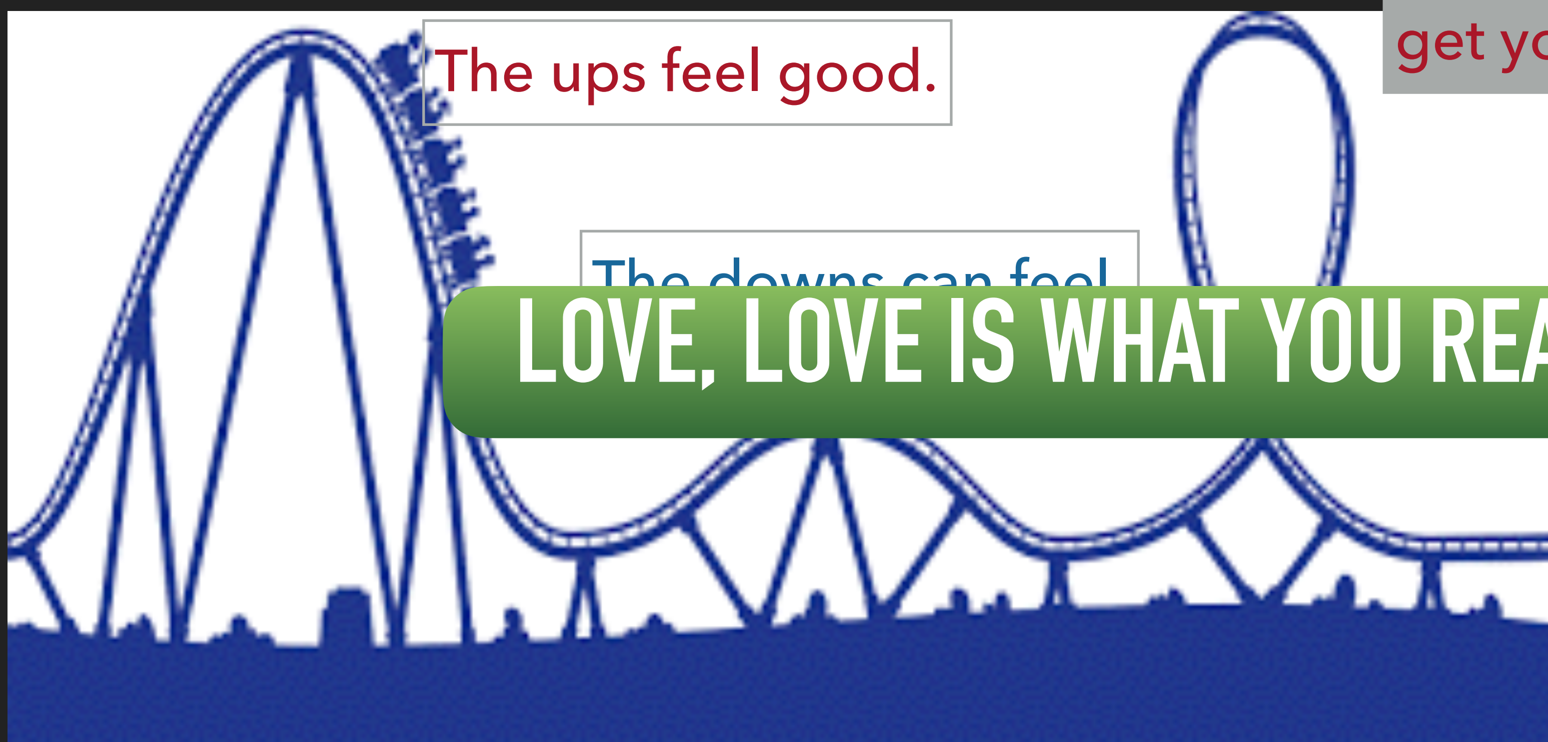
The downs can feel very down. Really.

The memories of “falling in love” can get you going again.

- And your advisor’s emotional wavelength matters, a lot.
- So choose your advisor well.

PhD research, like a relationship, has its ups and downs.

The memories of "falling in love" can get you going again.



The ups feel good.

The downs can feel

LOVE, LOVE IS WHAT YOU REALLY NEED.

- And your advisor's emotional wavelength matters, a lot.
- So choose your advisor well.

THE END

- Herrn G. Dopfer, my high school mathematics teacher, for encouraging me to not take English, focus on math and physics, and go to university, a first for our family
- Daniel Friedman, my advisor, for showing me what an advisor can do for a PhD student
- And two dozen PhD students, who had the guts to work with me and believed I could be their scientific and emotional guide

QUESTIONS?