G A D Ts

E N E R A L I Z E D

L G E B R A I C

A T A

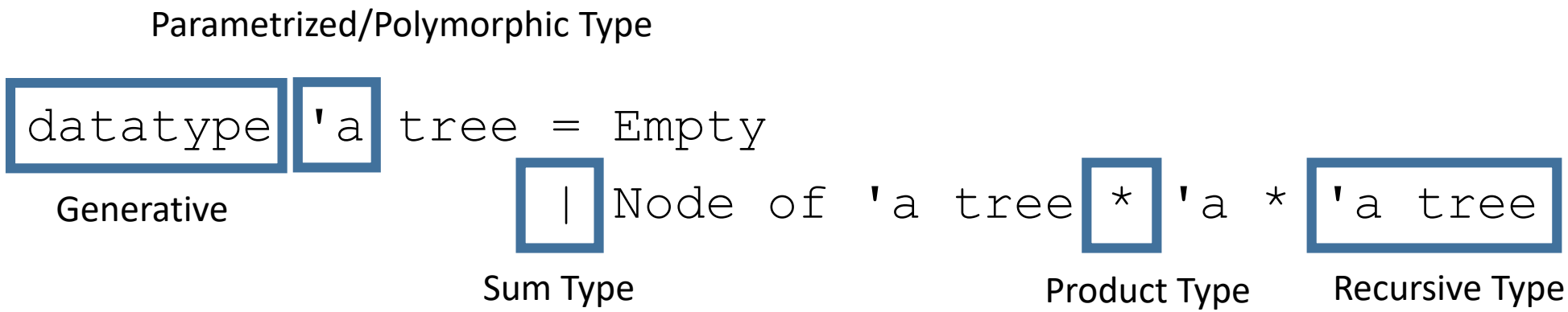
Y P E

“gadgets”

The humble datatype

```
type 'a tree = Empty
             | Node of 'a tree * 'a * 'a tree
```

The ~~humble~~ datatype



"Algebraic Datatype"

What are we generalizing?

Constructors!

Then the question is…

What are constructors?

Two things!

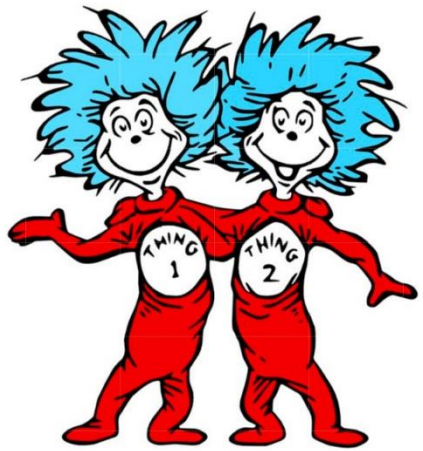
```
datatype 'a option = NONE | SOME of 'a
```

1.

```
NONE : 'a option
SOME : 'a -> 'a option
```

2.

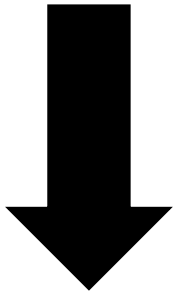
```
case e of
   NONE => …
 | SOME x => …
```



Generalized?

Step 1

```
datatype 'a option = NONE | SOME of 'a
```



Rewrite to reflect “Thing 1” that constructors are

```
datatype 'a option =
         NONE : 'a option
       | SOME : 'a -> 'a option
```

Then remove the requirement that

1. All constructors create values of the same type (sum)

2. All constructors that take in a polymorphic type return a polymorphic type (universal type)

1. *Not* all constructors create values of the same type

```
datatype 'a weird_option =
         NONE : string weird_option
       | SOME : int -> int weird_option
```

```
val x : int weird_option = SOME 10          ✓
val y : string weird_option = NONE          ✓
val x : string weird_option = SOME "hi"     X
```

2. *Not* all constructors that take in a polymorphic type return a polymorphic type

```
datatype 'a weirder_option =
         NONE : 'a weirder_option
       | SOME : 'a -> int weirder_option
```

```
val x : int weirder_option = SOME "weird"  ✓
val y : int weirder_option = SOME []       ✓
```

What do we have?

1. Generalized Sums

   Constructors can create values of different types

2. Existential Types

   Constructors that take in a polymorphic type do not have to return one

Generalized Sums

```
datatype 'a weird_option =
         NONE : string weird_option
       | SOME : int -> int weird_option
```

Generalized Sums

```
datatype 'a weird_option =
         NONE : string weird_option
       | SOME : int -> int weird_option
```

```
fn (e : 'a weird_option) =>
   case e of
      SOME x => x
    | NONE => 10
```

Could be `SOME` or `NONE`, since `'a` could be `int` or `string`

Generalized Sums

```
datatype 'a weird_option =
         NONE : string weird_option
       | SOME : int -> int weird_option
```

```
fn (e : int weird_option) =>
   case e of
      SOME x => x

fn (e : string weird_option) =>
   case e of
      NONE => "hello"
```

These are *exhaustive matches.*

Generalized Sums

```
datatype 'a weird_option =
         NONE : string weird_option
       | SOME : int -> int weird_option
```

```
fun wait_really (e : 'a weird_option) =
   case e of
      NONE => "hello"
    | SOME x => 10

wait_really : 'a weird_option -> 'a
```

This *typechecks.*

Let’s take a moment to appreciate this

Generalized Sums

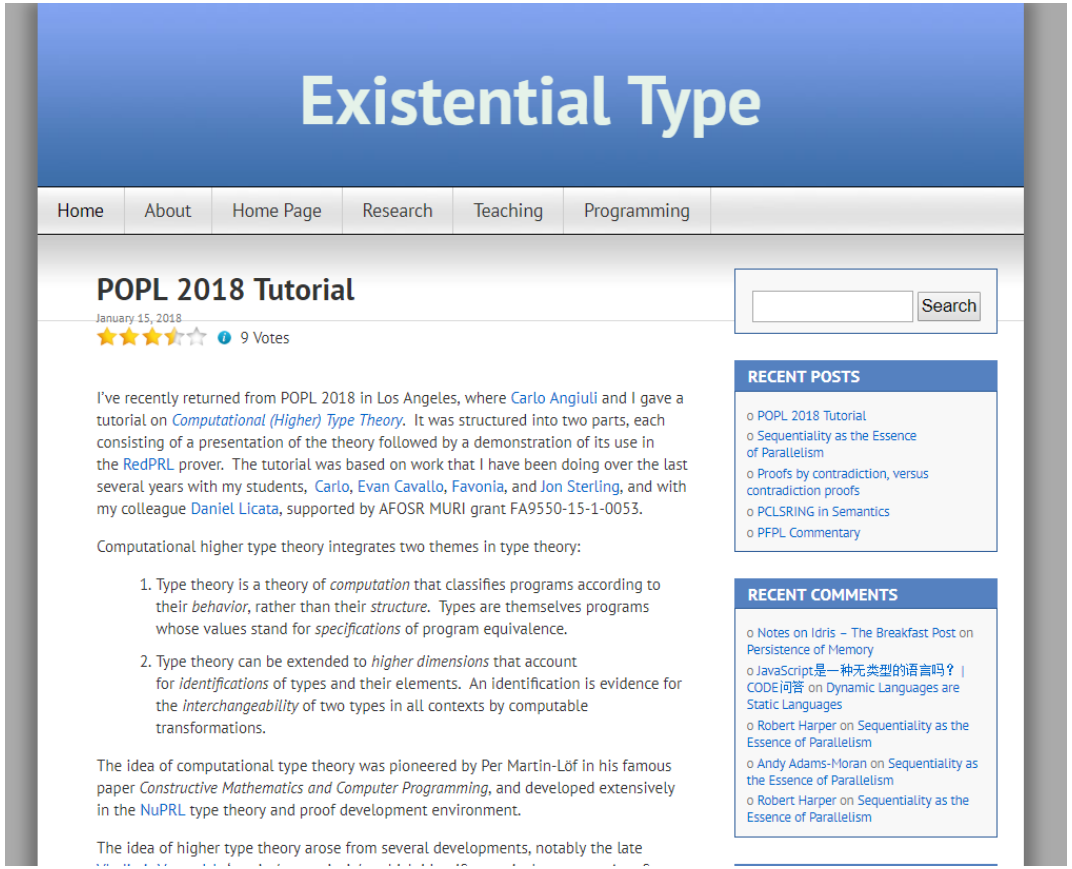
```
datatype 'a weird_option =
         NONE : string weird_option
       | SOME : int -> int weird_option
```

```
fun wait_really (e : 'a weird_option) =
    case e of
       NONE => "hello"
     | SOME x => 10

wait_really : 'a weird_option -> 'a
```

THIS TYPECHECKS

Existential Types



Existential Type

Home | About | Home Page | Research | Teaching | Programming

POPL 2018 Tutorial

January 15, 2018

9 Votes

I've recently returned from POPL 2018 in Los Angeles, where Carlo Angiuli and I gave a tutorial on *Computational (Higher) Type Theory*. It was structured into two parts, each consisting of a presentation of the theory followed by a demonstration of its use in the RedPRL prover. The tutorial was based on work that I have been doing over the last several years with my students, Carlo, Evan Cavallo, Favonia, and Jon Sterling, and with my colleague Daniel Licata, supported by AFOSR MURI grant FA9550-15-1-0053.

Computational higher type theory integrates two themes in type theory:

1. Type theory is a theory of *computation* that classifies programs according to their *behavior*, rather than their *structure*. Types are themselves programs whose values stand for *specifications* of program equivalence.
2. Type theory can be extended to *higher dimensions* that account for *identifications* of types and their elements. An identification is evidence for the *interchangeability* of two types in all contexts by computable transformations.

The idea of computational type theory was pioneered by Per Martin-Löf in his famous paper *Constructive Mathematics and Computer Programming*, and developed extensively in the NuPRL type theory and proof development environment.

The idea of higher type theory arose from several developments, notably the late

Search

RECENT POSTS

- POPL 2018 Tutorial
- Sequentiality as the Essence of Parallelism
- Proofs by contradiction, versus contradiction proofs
- PCLSRING in Semantics
- PFPL Commentary

RECENT COMMENTS

- Notes on Idris – The Breakfast Post on Persistence of Memory
- JavaScript是一种无类型的语言吗？ | CODE问答 on Dynamic Languages are Static Languages
- Robert Harper on Sequentiality as the Essence of Parallelism
- Andy Adams-Moran on Sequentiality as the Essence of Parallelism
- Robert Harper on Sequentiality as the Essence of Parallelism

Existential Types

```
datatype 'a weirder_option =
         NONE : 'a weirder_option
       | SOME : 'a -> int weirder_option
```

Type theory break

Existential Types

```
datatype 'a bobs_blog =
         NONE : 'a bobs_blog
       | SOME : ('a * 'a -> bool) -> int bobs_blog
```

Existential Types

```
datatype 'a existential =
         NONE : 'a weirder_option
       | SOME : ('a * 'a -> bool)
                -> int weirder_option
```

```
fn (e : int existential) =>
   case e of
      NONE => true
    | SOME 'a (x, f) => f x
```

Ocaml Demo