Type Theory in Software Verification

TYPES in Munich

Gidon Ernst
gidon.ernst@lmu.de LMU Munich

This Talk

- Question:

  What's the impact of research in Type Theory on "practical" Software Verification

a (major) side interest

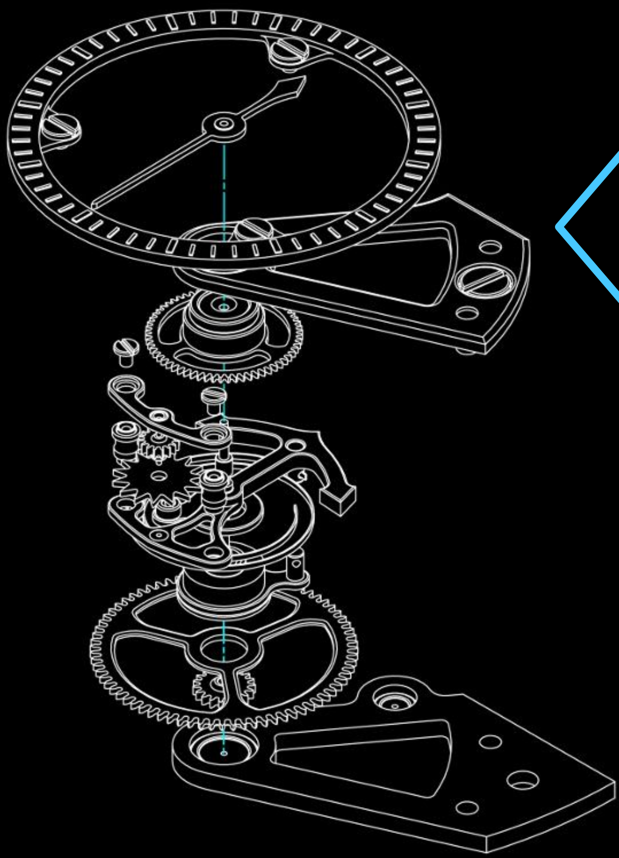
my main research area

- Background
- Examples
- Conclusions

Software Verification (Analogy)

concrete system

abstract model



laws of physics

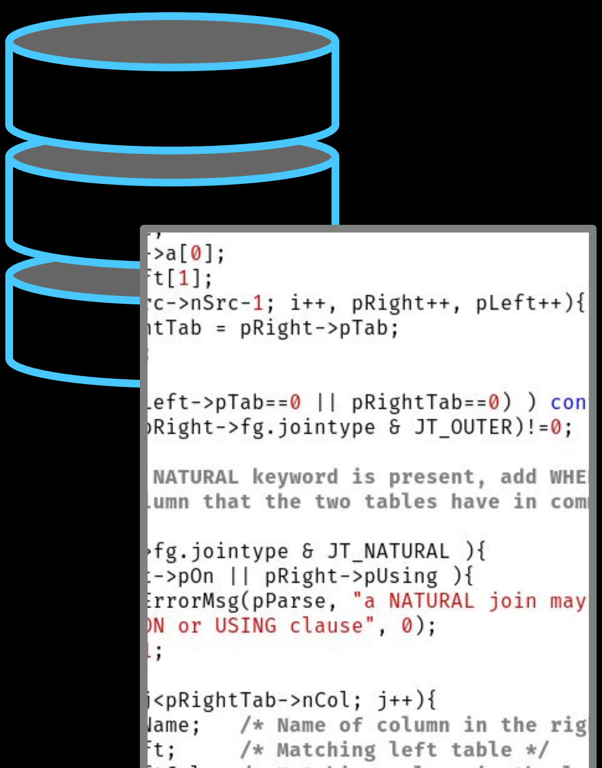
$$\frac{d\,hour}{d\,minute} = \frac{1}{60}$$

easy to understand
unambiguous

Software Verification

concrete system

abstract model



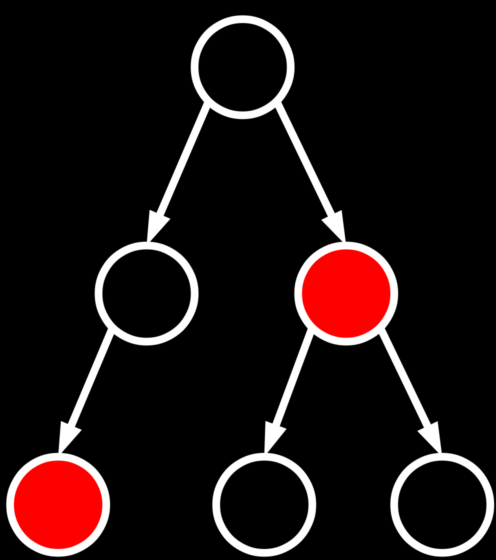
laws of programming

read(write(x)) = x

hard with automation
impossible without

How much Effort?

standard algorithms: > 1 h	academic prototype: 1-5 py	production quality: > 20 py
search tree	file systems	CompCert C compiler
		OS Kernel



How much Effort?

standard algorithms: > 1 h

academic prototype: 1-5 py

production quality: > 20 py

bottlenecks:

writing good specifications
engineering & tool aspects
new correctness criteria
proof automation

Typical Hoare-style verification (Dafny)

```
method bsearch(value: int, a: array<int>) returns (res: bool)
    requires a ≠ null && sorted(a)
    ensures  res == contains(value, a)
{
    var low, high := 0, a.Length;
    while low < high
      invariant forall i ::
         0 ≤ i < low || high ≤ i < a.Length ⟹ a[i] ≠ value
   {
      ...
   }
   return false;
}
```

specification

proof guidance

https://rise4fun.com/dafny/tutorialcontent/guide#h211

Type Theory (for the purpose of this talk)

- Expressiveness: proposition = type
  - $Even = \{ n \mid n \,\%\, 2 = 0 \}$
  - $x : Vector\ n$

ease of specification

- Constructivism: proof = program
  - $solve : A \longrightarrow Bool$ (terminates)
  - $com : n \longrightarrow m \longrightarrow n + m \equiv m + n$
    (lemmas = functions)

ease of proof

Automath [de Bruijn 1967]

- early general & useful proof checker
- based on type theory
  (flexible choice of which kind)
- high influence on design of later tools

Coq

- Calculus of inductive constructions
  [Coquand, Huet, 1988]
- Impressive applications to software development
  - CompCert, DeepSpec, FSCQ, …
- Similarly: verified programming in
  - Agda, Idris, Epigram, Lean (mostly math)
  - F★ (verified crypto in Firefox!)

PVS [Owre, Shankar, Rushby 1992]
classical, predicative + dependent types

```
below(i): TYPE
  = {s: nat | s < i}
```

~ set-based semantics

```
is_finite(s): bool
  = (EXISTS n,
          (f: [(s) → below[n]]): injective?(f))
```

```
finite_set: NONEMPTY_TYPE
  = (is_finite) CONTAINING emptyset
```

Data Invariants (Why3, VCC, JML, ...)

```
type array 'a = {
  elts   : int ⟶ 'a;
  length : int
} invariant {
  0 ⩽ length
}
```

- Invariants are re-checked after modifications

Lemma Functions (Dafny, Why3, VCC, …)

```
function index(x: T, xs: seq<T>) returns (r: int)
  decreases |xs|  // inductive measure
  ensures r ≥ 0 ⟹ contains(x, xs) && xs[r] == x
{
  if xs == [] { r := -1; }
  else if x == xs[0] { r := 0; }
  else { r := index(xs[1..]); r := r + 1; }
}
```

programs represent proofs

Synthesis with Refinement Types [Polikarpova et al 2016]

```
data BST a where
  Empty :: BST a
  Node  :: x: a → l: BST {a | _v < x}
                → r: BST {a | _v > x} → BST a

measure keys :: BST a → Set a where
  Empty → []
  Node x l r → keys l + keys r + [x]

insert :: x: a → t: BST a
              → {BST a | keys _v == keys t + [x]}
insert = ??
```

search guided by type structure

New theories for Program Verification

- Mechanize meta-theory in proof assistant
- Two alternatives:
  - implement tool based on that (e.g. VeriFast)
    $\longrightarrow$ potential gain in automation
  - shallow embedding into common logic (e.g. Coq)
    $\longrightarrow$ re-use existing infrastructure

both approaches are practical and successful

Type Theory: Criticism
(Context: QED Manifesto retrospective)

Instead of trying to prove *as many* true statements as possible, constructive mathematics is about making it *difficult* to prove something. (Of course, *if* you then prove it, the proof contains a bit more information.)

[Wiedijk 2007]

The HOL type system is too poor. As we already argued in the previous section, it is too weak to properly do abstract algebra.

my opinion: theory is not at fault but user interface

Take-Away

- "Pure" Type-Theory
  - elegantly captures key concepts
  - good as foundations
  - good as vehicle of thought

- "Messy" verification methodology for programs
  - needs to cope with practical issues
  - focus on efficient and effective automation
  - gains a lot by incorporating foundational concepts