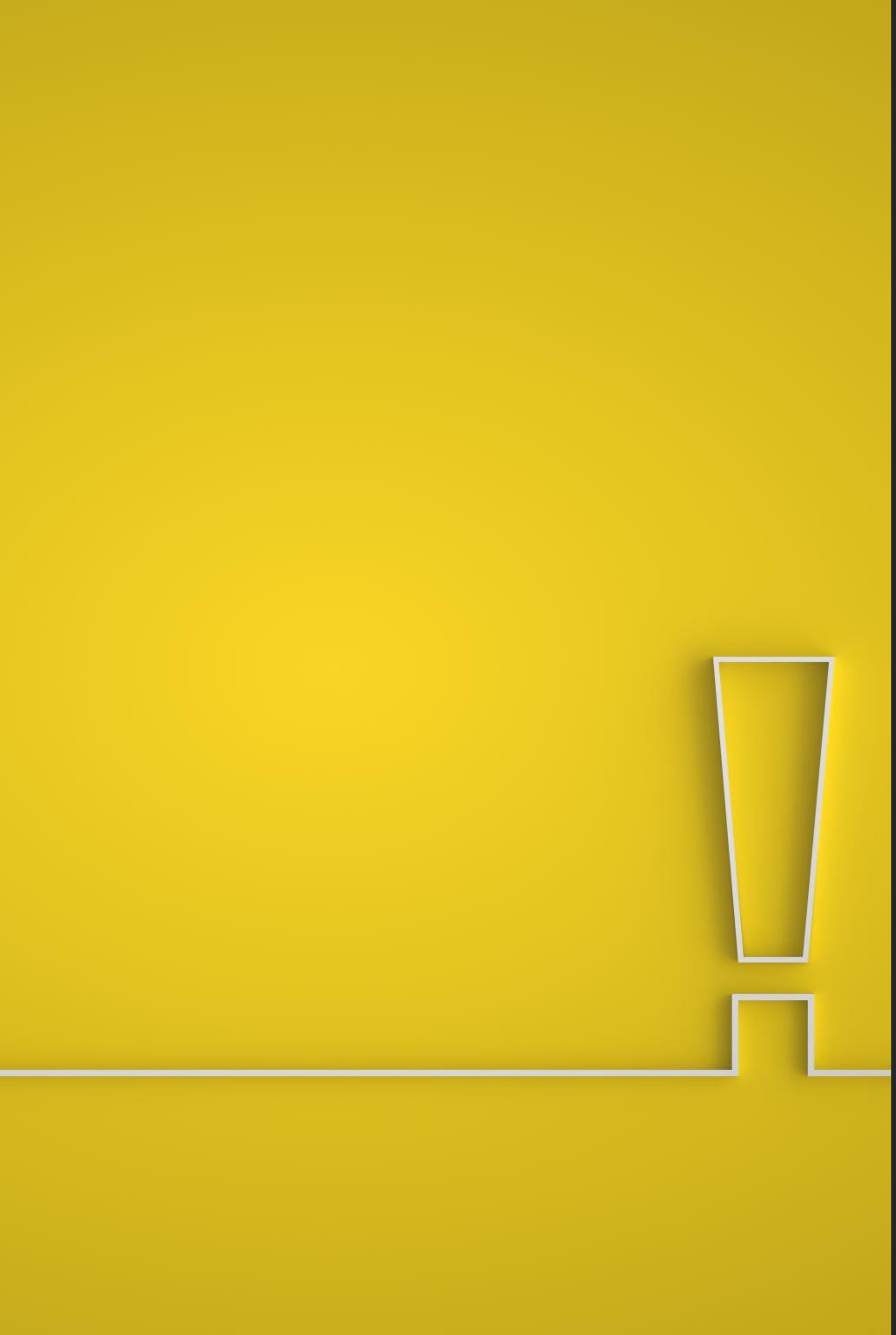
Typed Assembly Languages

ROSS TATE



Goal

To have existing compilers for multiple major GCed languages be able to generate WebAssembly with minimal modifications and without the host needing to trust the application to be memory safe.

Memory-Safety Mechanisms

DYNAMIC

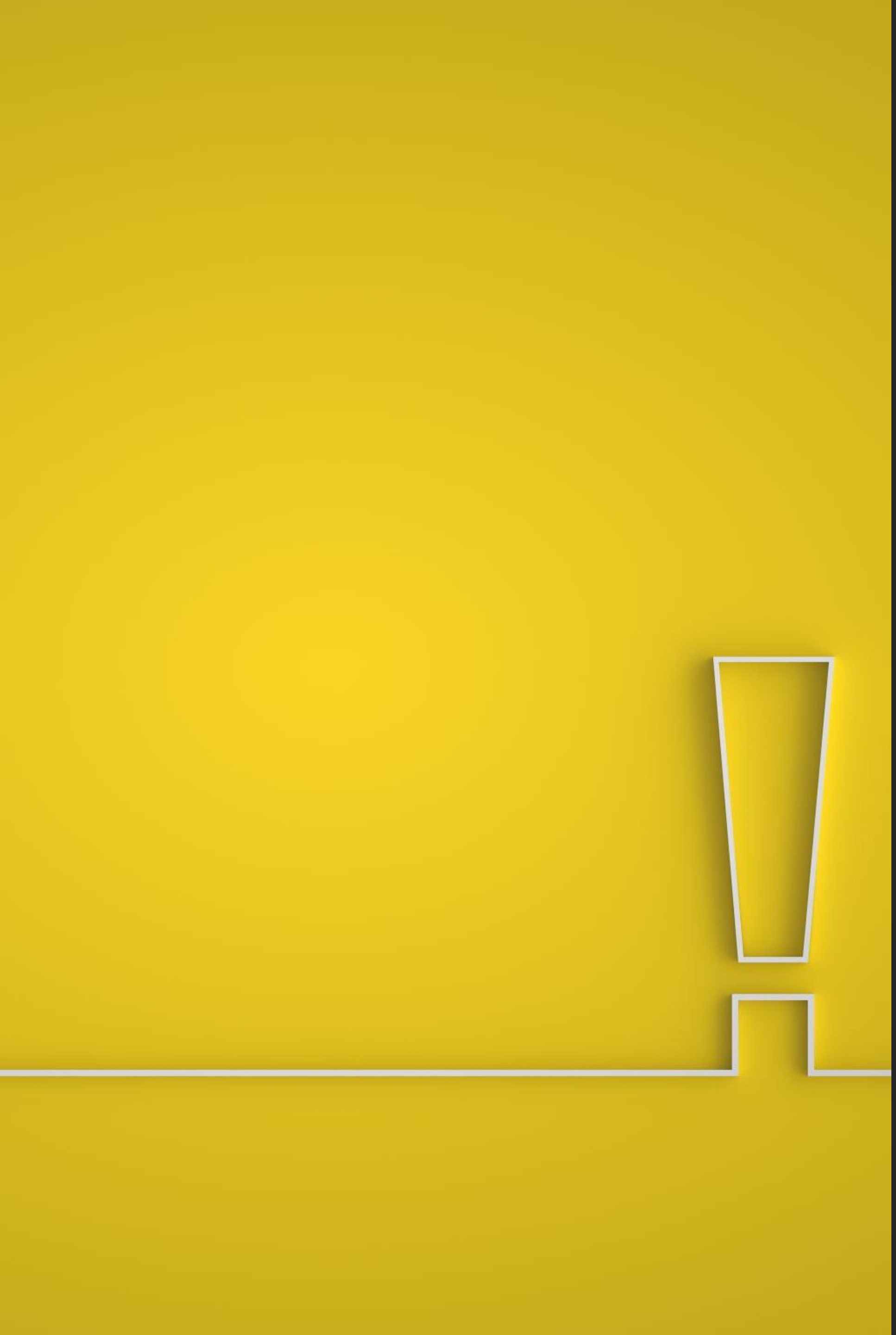
Casts

Bounds Checks

STATIC

Types

Proofs



Research Goal

To have existing compilers for multiple major GCed languages be able to generate typed assembly with minimal modifications such that the type system is reliably checkable and ensures safety.

Design Challenges

Design a *sound* type system that

1. Can express the relevant invariants
2. Is reliably checkable
3. Is practical to generate

The Golden Age

THE LATE ’90S

The Golden Age

Laid the foundations for
Typed Assembly Languages (TALs)

Identified first major challenges
and initial solution strategies

Compiled an expressive academic
language to an independently
verifiable executable format

Typed Closure Conversion

YASUHIKO MINAMIDE GREG MORRISETT ROBERT HARPER

Closures

sum(x, y : int) : () -> int

= λ(). x + y

lookup(a : int[], i : int) : () -> int

= λ(). a[i]

A () -> int is a pair of

- a capture of the environment
- a code pointer expecting that capture as an argument

Ideals vs. Trust in Low-Level Code

```
i32 good_apply(t : IntThunk) {
    env = t.fst;
    code = t.snd;
    return code(env);
}
```

```
i32 bad_apply(t1, t2 : IntThunk) {
    env1 = t1.fst;
    code2 = t2.snd;
    return code2(env1);
}
```

Attacking Misplaced Trust

sum(x, y : int) : () -> int

= λ(). x + y

lookup(a : int[], i : int) : () -> int

= λ(). a[i]

```
i32 bad_apply(t1, t2 : IntThunk) {
    env1 = t1.fst;
    code2 = t2.snd;
    return code2(env1);
}
```

Accesses arbitrary memory

How to support good and prevent bad?

Existential Types

```
i32 good_apply(t : IntThunk) {
    env = t.fst;
    code = t.snd;
    return code(env);
}
```

code and env have same α

```
i32 bad_apply(t1, t2 : IntThunk) {
    env1 = t1.fst;
    code2 = t2.snd;
    return code2(env1);
}
```

code2 and env1 have different α

type IntThunk = ∃α. Pair<α, CodePtr(α → i32)>

All MVP Proposals

```
i32 good_apply(t : IntThunk) {
    env = t.fst;
    code = t.snd;
    return code(env);
}
```

```
i32 bad_apply(t1, t2 : IntThunk) {
    env1 = t1.fst;
    code2 = t2.snd;
    return code2(env1);
}
```

Casts the anyref

type IntThunk = Pair<anyref, CodePtr(anyref ⟶ i32)>

Takeaway

WebAssembly will need existential quantification to eliminate superfluous casts on closure calls, virtual-method calls, interface-method calls, and so on.

From System F to Typed Assembly Language

GREG MORRISETT
DAVID WALKER
KARL CRARY
NEAL GLEW

Pseudo-Instructions

```
i32 good_apply(t : IntThunk) {
    [β, p] = unpack t;
    β env = p.fst;
    Code(β → i32) code = p.snd;
    return code(env);
}
```

```
IntThunk thunk(x : i32) {
    Pair<i32, Code(i32 → i32)> p
        = new Pair<...>(x, body);
    return pack [i32, p] as IntThunk;
}

i32 body(x : i32) { return x; }
```

type IntThunk = ∃α. Pair<α, Code(α → i32)>

Stack-Based Typed Assembly Language

GREG MORRISETT
KARL CRARY
NEAL GLEW
DAVID WALKER

Stack Typing

- Custom Calling Conventions
- (Multiple) Return Addresses
- Stack-Allocated Data

TALx86: A Realistic Typed Assembly Language

GREG MORRISETT
KARL CRARY
NEAL GLEW
DAN GROSSMAN
RICHARD SAMUELS
FREDERICK SMITH
DAVID WALKER
STEPHANIE WEIRICH
STEVE ZDANCEWIC

Typestate for Memory Allocation

malloc<τ> returns a reference to τ where all fields are uninitialized

Write to a field updates initialization state of that reference’s type

Handles immutable fields and field types without default values

Takeaway

WebAssembly will need
typestate or the like
to support low-level initialization,
especially for “immutable” fields
or types without default values.

Scalable Certification for Typed Assembly Language

DAN GROSSMAN
GREG MORRISETT

Experimental Results

Annotations introduced roughly 50% overhead to code size

- Already employing techniques in current MVP

Validation time roughly 50% relative to compilation time

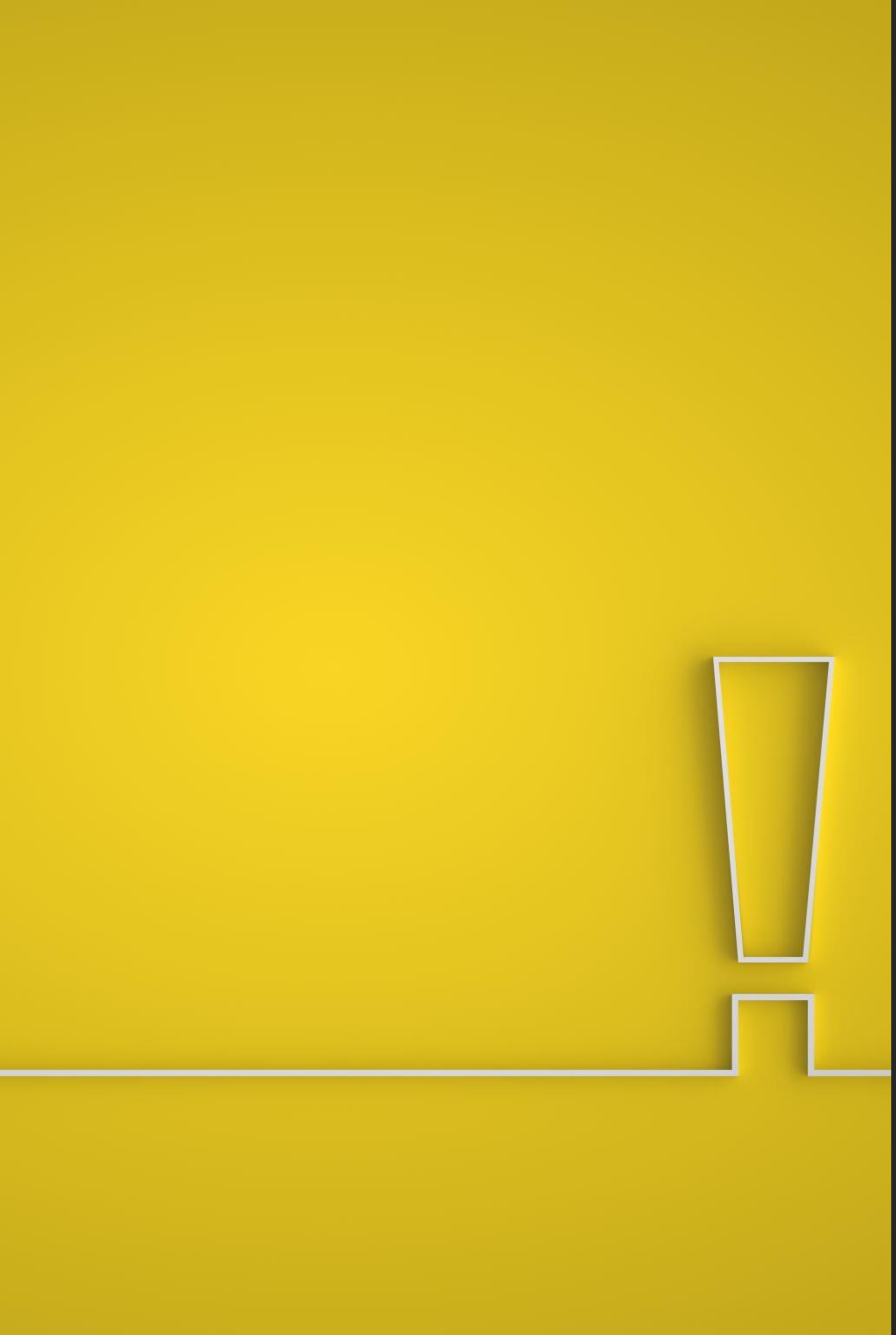
- More efficient type-checking algorithm than current MVP

Explored annotation-size vs. validation-time tradeoff

- Eliminating input types on non-merging blocks reduced *both* annotation size *and* validation time by roughly 15% each

Takeaway

Eliminating unnecessary type annotations makes meaningful improvements to annotation size and validation time.



Research Goal

To have a ~~existing~~ compiler~~s~~ for a ~~multiple major~~ GCed language~~s~~ be able to generate typed assembly ~~with minimal modifications~~ such that the type system is reliably checkable and ensures safety.

The Age of Exploration

THE EARLY ’00S

The Age of Exploration

- Goal: Compile a major language (e.g. Java) to an independently verifiable executable format
- Explored many interesting ideas and extensions to prior methods
- Kept running into problems and making major concessions

Obstacles

SUBTYPING

Bounded polymorphism is undecidable

- Mitigatable through pseudo-instructions
- Surface-level subtypes no longer map to low-level subtypes
- Cannot take advantage of variance

CASTING

Branching informs types

- Special-casing can be done for sum types
- And more generally for RTT tag hierarchies
- Cannot reason about generics with variance

How to cast Object[] to Foo[] and know all elements are Foo values without trust?

Takeaway

Expressing the invariants of language implementations is the primary challenge.

The Industrial Age

THE LATE ’00S

The Industrial Age

- Reexamined foundations for typed assembly languages
- Identified new high-level methods for expressing program invariants
- Successfully modified existing compiler for a major language to produce verifiable executables

A Simple
Typed Intermediate Language
for Object-Oriented Languages

JUAN CHEN

DAVID TARDITI

Quantify over Domain-Specific Abstractions

Rather than having α in $\exists\alpha$ abstract low-level types, have α in $\exists\alpha$ abstract language-specific constructs

- Such as classes in OO languages
- Or structural types in functional languages

Predicativity helps address decidability

Name Runtime Structures

Instance(α)

- vtable : Ref(VTable(α))
- …

VTable(α)

- id : Identifier(α)
- super : $\exists\beta \gg \alpha.\ \mathrm{Ref}_N(\mathrm{VTable}(\beta))$
- hashcode : $\mathrm{CodePtr}((\exists\beta \ll \alpha.\ \mathrm{Ref}(\mathrm{Instance}(\beta))) \rightarrow \mathrm{i32})$
- …

Refine using Partial Information

Instance($\alpha \ll$ AbstractList)

- vtable : Ref(VTable(α))
- modCount : mut i32
- ...

Instance($\alpha \ll$ ArrayList)

- vtable : Ref(VTable(α))
- modCount : mut i32
- size : mut i32
- elems : mut $\exists\beta.\ \mathrm{Ref}_N(\mathrm{Array}(\beta))$
- ...

Check that more informative bounds map to more precise structures

Custom Casting with Identifiers

$r1 : \exists\alpha.\ \mathrm{Ref}(\mathrm{Array}(\alpha))$

open r1 as β ;; $r1 : \mathrm{Ref}(\mathrm{Array}(\beta))$

r2 := r1.elemType ;; $r2 : \mathrm{Identifier}(\beta)$

r3 := global_id_for_String

br_ne r2 r3 $cast_failed

;; β = String because Identifiers are equal

;; r1 now has type Ref(Array(String))

Array(α)

- elemType : Identifier(α)
- elems : $\mathrm{array}(\exists\beta \ll \alpha.\ \mathrm{Ref}_N(\mathrm{Instance}(\beta)))$

global_id_for_String : Identifier(String)

Nominal reasoning

Takeaway

Nominal types better express the invariants that are important to major programming languages and common runtimes.

Type-Preserving Compilation for Large-Scale Optimizing Object-Oriented Compilers

JUAN CHEN CHRIS HAWBLITZEL FRANCES PERRY MIKE EMMI
JEREMY CONDIT DERRICK COETZEE POLYVIOS PRATIKAKIS

Compiling C# 1.0 to Typed x86

Annotations introduced roughly 105% overhead to code size

- Including guarantee that all array accesses were in bounds
- Later improved to roughly 9%

Validation time roughly 6% relative to compilation time

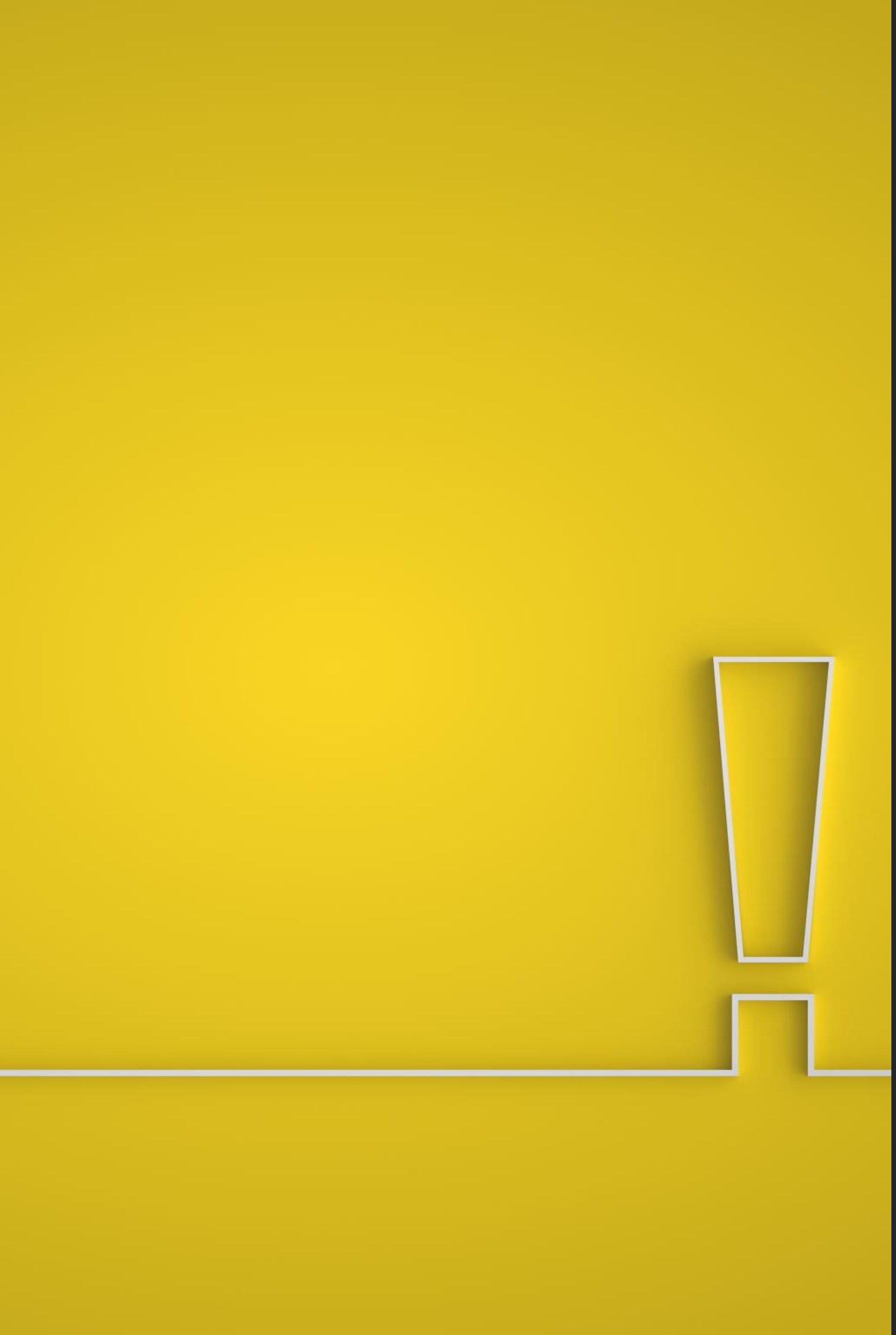
- Later improved to roughly 3%

Modified roughly 10% (19K of 200K LOC) of the existing compiler

Takeaway

Nominal types are more compact and efficiently checkable.

Generation burden for eliminating superfluous casts can be substantial.



Research Goal

To have an ~~existing~~ compiler~~s~~ for a ~~multiple~~ ~~major~~ GCed language~~s~~ be able to generate typed assembly ~~with minimal modifications~~ such that the type system is reliably checkable and ensures safety.

Inferable Object-Oriented Typed Assembly Language

ROSS TATE JUAN CHEN CHRIS HAWBLITZEL

Framework for Existential Types

Decidable Subtyping

Computable Joins

Complete
Loop-Invariant Inference

No
pseudo-instructions
necessary!

Compiling C# 1.0 to iTalX86

Annotations introduced roughly 4% overhead to code size

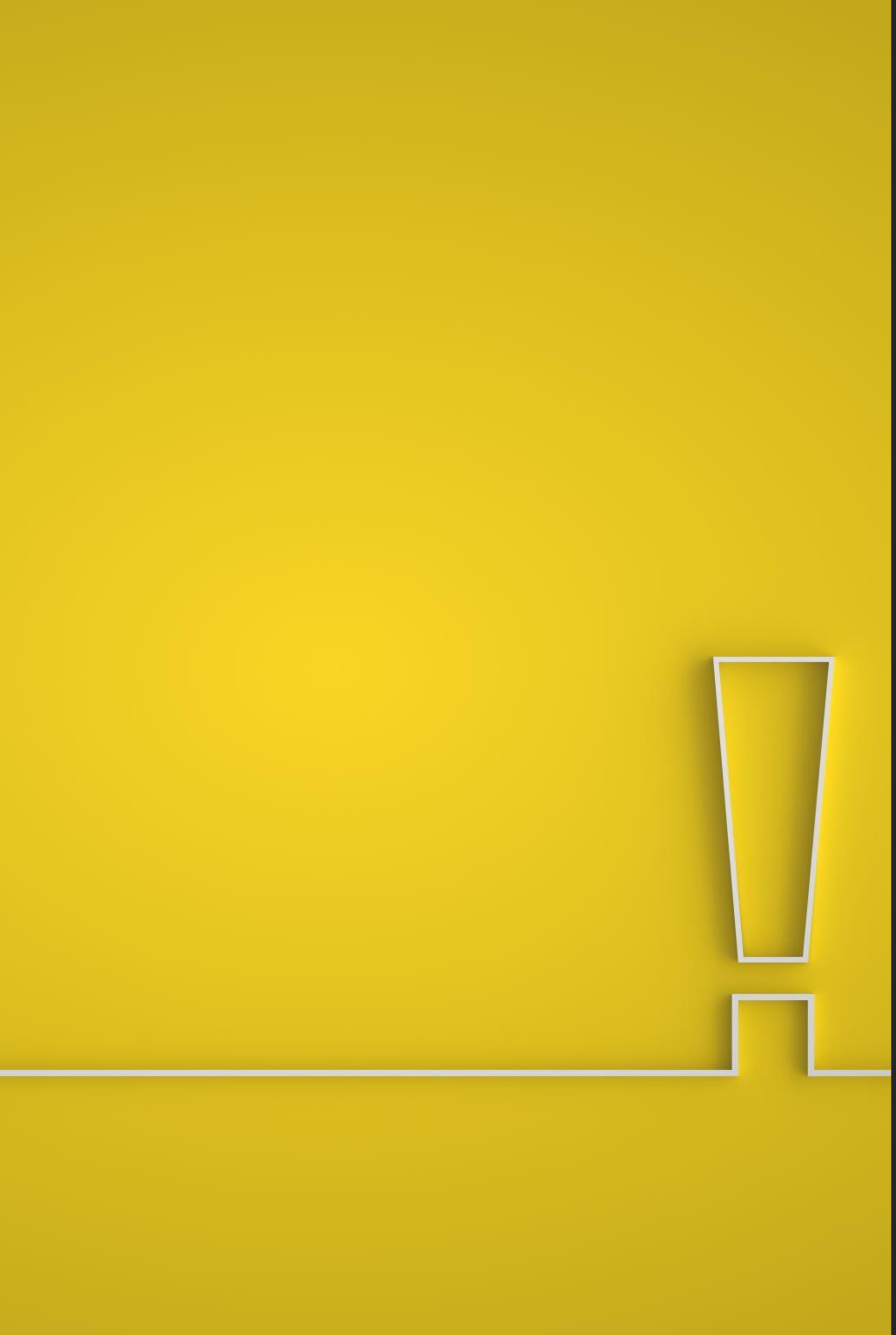
- Including guarantee that all array accesses were in bounds

Validation time roughly 8% relative to compilation time

- Including loop-invariant inference time

Modified roughly 2.5% (5K of 200K LOC) of the existing compiler

- Only 0.5K were changes to existing code
- Remaining 4.5K was just new code for outputting meta-information



Research Goal

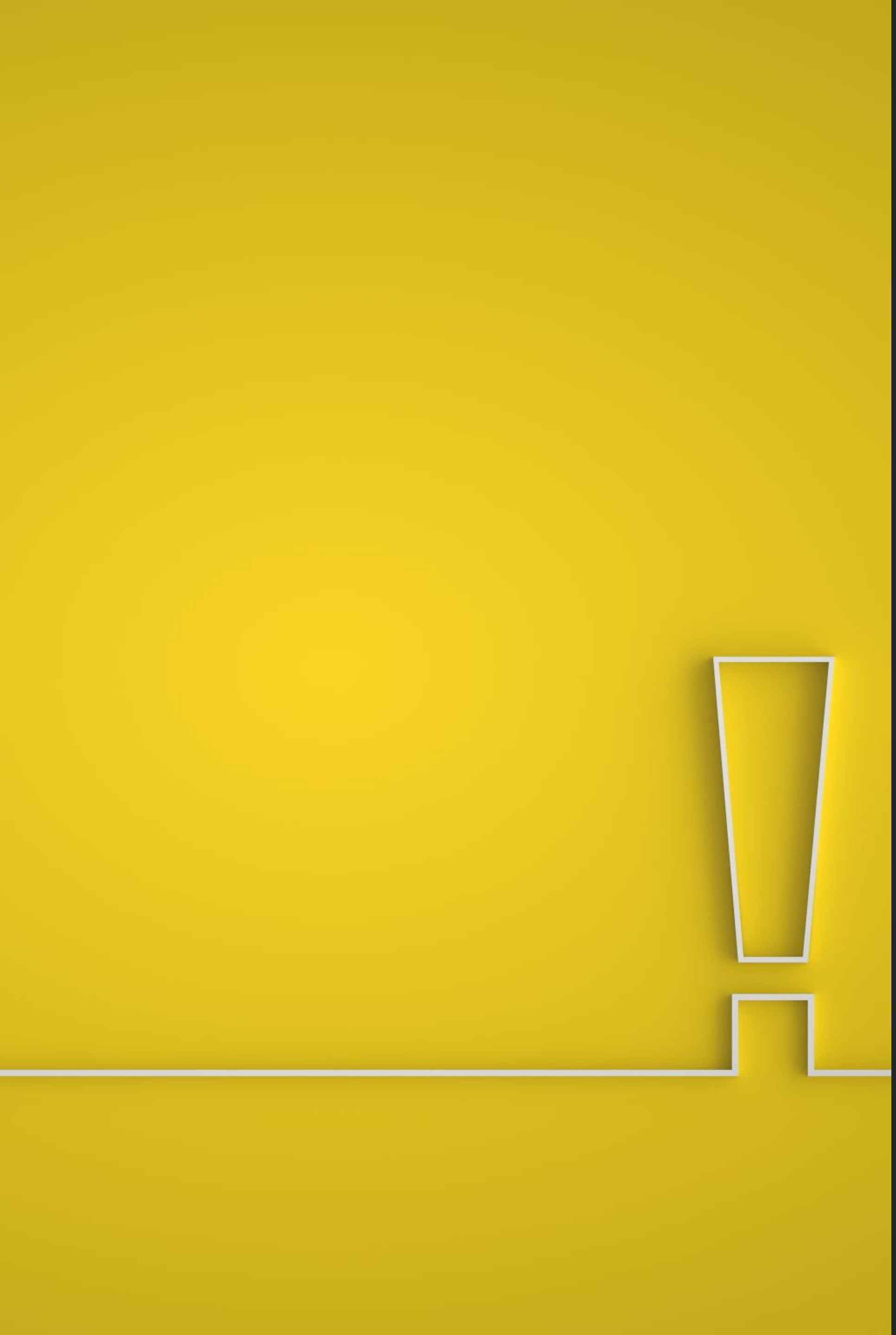
To have an existing compiler~~s~~ for a ~~multiple~~ major GCed language~~s~~ be able to generate typed assembly ~~with minimal modifications~~ such that the type system is reliably checkable and ensures safety.

Takeaway

Careful design of existential types can substantially reduce generation burden.

The Information Age

NOW



Goal

To have an ~~existing~~ compiler~~s~~ for a ~~multiple~~ ~~major~~ GCed language~~s~~ be able to generate WebAssembly ~~with minimal modifications~~ and without the host needing to trust the application to be memory safe.