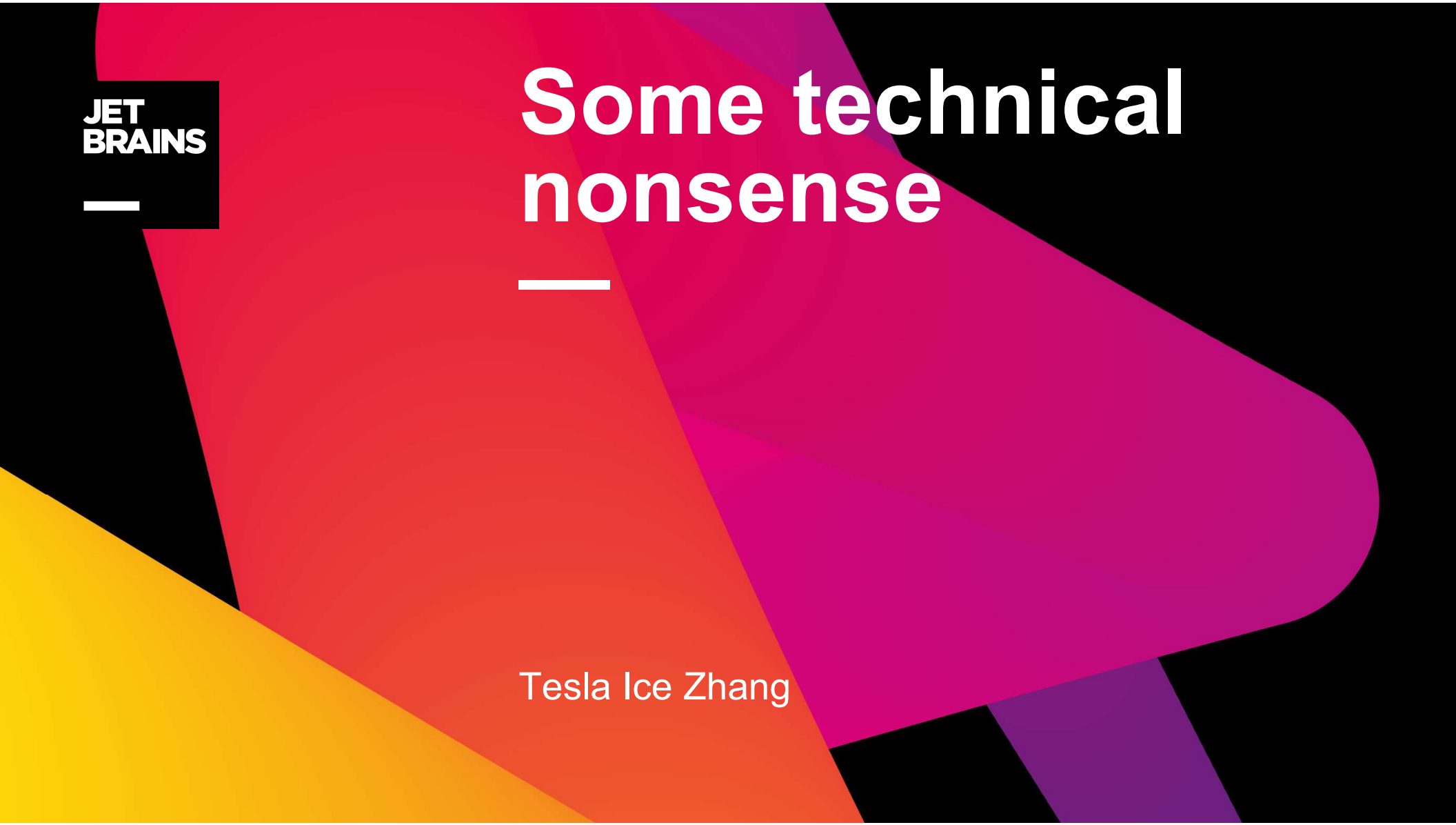

JET
BRAINS
Some technical
nonsense
Tesla Ice Zhang

About me

Undergraduate sophomore at Penn State, 2 years of experience in SDE intern

- Sourcebrella (IDE plugin & frontend), PingCAP (TiKV, gRPC, protobuf), JetBrains Research (Arend & intellij-arend & arend-lib)
- IDE & editor development, compiler & typechecker (mostly DTLC)
- (Homotopy) type theory and its constructive interpretations
- Interested in making friends

I want to

—

… talk about Arend, the project I was working on in JetBrains Research, along with some minor discovery in IDE architecture design.

Arend

… is a programming language & proof assistant based on HoTT-I. It has usual DT+HoTT features such as TDD, class system, V type & coe & Path & HITs. Apart from that, there's also Java FFI (external tactics), decent IDE support (IntelliJ IDEA), syntactic homotopy truncation, and a controversial syntax.

https://arend-lang.github.io/

Equation reasoning

```
\import Arith.Nat
\open Nat
\open NatSemiring

\func *-comm (n m : Nat) : n * m = m * n
  | 0,     0     => idp
  | suc n, 0     => *-comm n 0
  | 0,     suc m => *-comm 0 m
  | suc n, suc m => pmap suc (
    suc n * m + n   ==< pmap (`+ n) (*-comm _ _)                 >==
    m * n + m + n   ==< +-assoc _ _ _                            >==
    m * n + (m + n) ==< pmap2 (+) (inv (*-comm n m)) (+-comm m n) >==
    n * m + (n + m) ==< inv (+-assoc _ _ _)                      >==
    n * m + n + m   ==< pmap (`+ m) (*-comm _ _)                 >==
    suc m * n + m   `qed)
```

Higher inductive types

—

```
\data Torus
  | point
  | line1  I \with { left => point | right => point }
  | line2  I \with { left => point | right => point }
  | face I I \with {
    | left,  i => line2 i
    | right, i => line2 i
    | i, left  => line1 i
    | i, right => line1 i
  }
```

```
\data Sphere1
  | base1
  | loop I \with {
    | left  => base1
    | right => base1
  }
```

Syntactic homotopy truncation

—

```
\truncated \data Quotient {A : \Type} (R : A -> A -> \Type) : \Set
  | in~ A
  | ~-equiv (x y : A) (R x y) (i : I) \elim i {
    | left  => in~ x
    | right => in~ y
  }
```

Limitations

- Univalence/coe doesn’t compute (lacks Glue/hcomp in CTT, V/hcom in CCCTT), therefore no canonicity
- Side effects in Java FFI are not tracked
- No string/char type
- Truncated paths are directly erased – erasure under a proof-relevant system

IDE features

https://arend-lang.github.io/about/intellij-features

The expression problem

… is a famous software-engineering problem solved by design patterns (Object Algebra and Tagless Final). I need it as a prerequisite of a simple discovery in IDE design.

Tagged union

Assume an enum (or a datatype with multiple variants in Haskell) and a function over it.

```
#[derive(Copy, Clone)]
enum A { B, C }

fn f(a: A) {
    match a {
        A::B => {…},
        A::C => {…},
    }
}
```

Tagged union

Adding a function over the enum won’t affect existing code.

We may publish A and f as a library, and users can extend it with g.

```
#[derive(Copy, Clone)]
enum A { B, C }

fn f(a: A) {
    match a {
        A::B => {…},
        A::C => {…},
    }
}

fn g(a: A, b: A) {
    match (a, b) {
        (A::B, A::B) => {…},
        (A::B, A::C) => {…},
        (A::C, b) => {…},
    }
}
```

Tagged union

Adding an enum variant to A requires modifying all functions over A.

If we publish A and f as a library, it's not possible to extend it with A::D without modifying the library itself.

However, we can easily extend all functions with the additional support for A::D with the IDE.

```
#[derive(Copy, Clone)]
enum A { B, C, D, }

fn f(a: A) {
    match a {
        A::B => {…},
        A::C => {…},
        A::D => {…},
    }
}
fn g(a: A, b: A) {
    match (a, b) {
        (A::B, A::B) => {…},
        (A::B, A::C) => {…},
        (A::B, A::D) => {…},
        (A::C, b) => {…},
        (A::D, b) => {…},
    }
}
```

Tagged union

Prevent extension from breaking code: catch-all patterns.

```
#[derive(Copy, Clone)]
enum A { B, C, D, }

fn f(a: A) {
    match a {
        A::B => {…},
        A::C => {…},
        _ => {…},
    }
}
fn g(a: A, b: A) {
    match (a, b) {
        (A::B, A::B) => {…},
        (A::B, A::C) => {…},
        (A::B, _) => {…},
        (A::C, b) => {…},
        (_, b) => {…},
    }
}
```

Subtyping polymorphism

Assume an abstract class with some subclasses with an abstract method.

We can see the method as a function over A, while B and C are variants of A.

```
abstract class A {
  abstract int f();
}
class B extends A {
  @Override int f() { return 114514; }
}
class C extends A {
  @Override int f() { return 1919810; }
}
```

Subtyping polymorphism

Adding a variant is easy. No need to touch existing codebase.

```
abstract class A {
  abstract int f();
}
class B extends A {
  @Override int f() { return 114514; }
}
class C extends A {
  @Override int f() { return 1919810; }
}
class D extends A {
  @Override int f() { return 1919810; }
}
```

Subtyping polymorphism

Adding a method requires modifying all subclasses.

However, we can easily extend all subclasses with the additional support for A::g with the IDE.

```
abstract class A {
  abstract int f();
  abstract String g();
}
class B extends A {
  @Override int f() { return 114514; }
  @Override String g() { return "Boy"; }
}
class C extends A {
  @Override int f() { return 1919810; }
  @Override String g() { return "Next"; }
}
class D extends A {
  @Override int f() { return 1919810; }
  @Override String g() { return "Door"; }
}
```

Subtyping polymorphism

Prevent extension from breaking code: default implementation.

```
abstract class A {
  abstract int f();
  String g() { return "That’s good"; }
}
class B extends A {
  @Override int f() { return 114514; }
}
class C extends A {
  @Override int f() { return 1919810; }
}
class D extends A {
  @Override int f() { return 1919810; }
}
```

Summary

Extension to a codebase is bidirectional – more variants v.s. more functions.

Tagged union & subtyping polymorphism are extensible at either side, but not the other side. Both are capable to prevent breakage.

IDEs

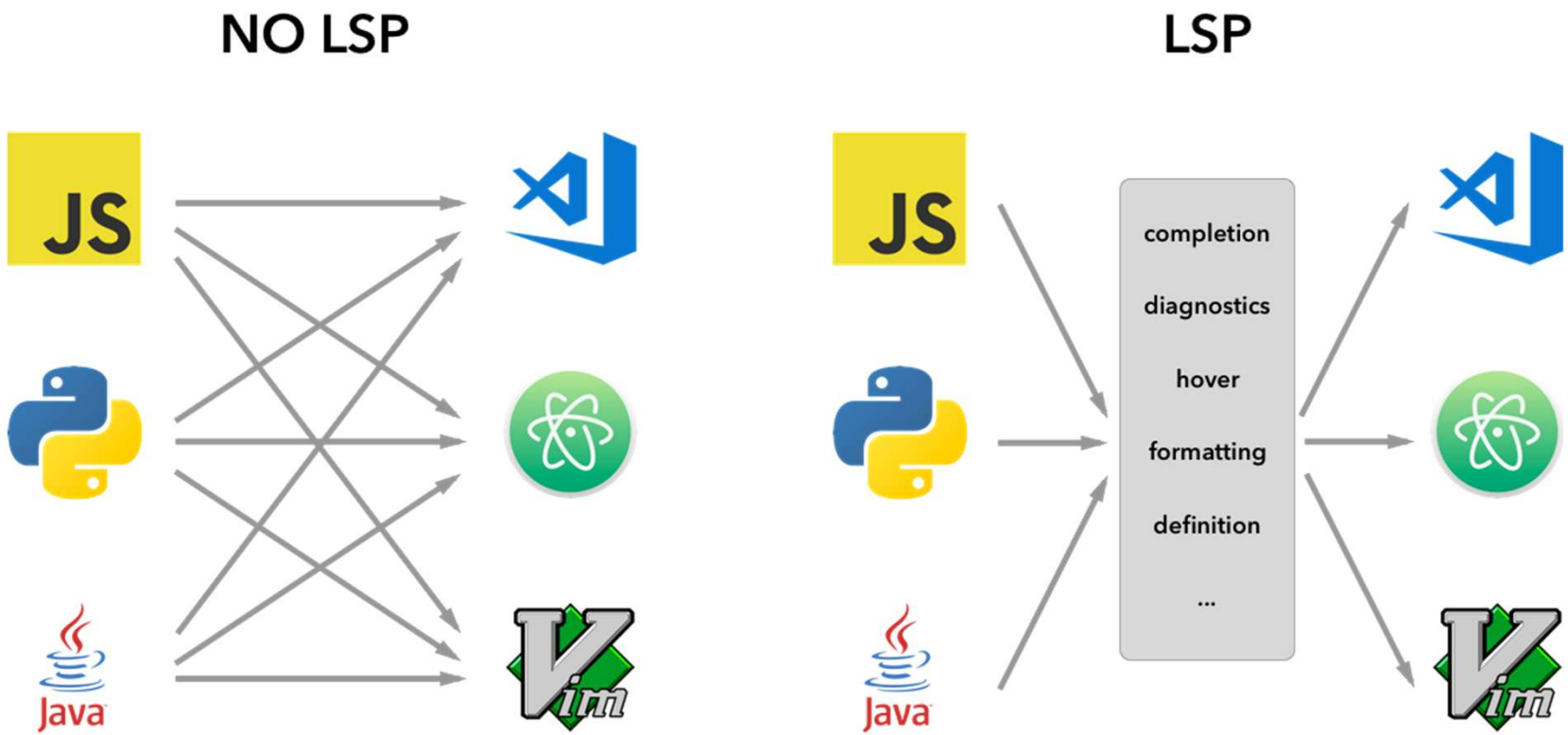
Now, let’s look at IDEs.

Lightweight code editors can *become* an IDE with **language servers** installed.
I’m going to talk about IDEs like this.

LSP – the concept

Editor developers focus on the editor part
Language developers focus on the language support

LSP – the concept



LSP – flawed

It looks very nice by the first glance. However, there's a big problem – interaction among multiple plugins is almost impossible.

Consider a mixed Java-Kotlin project. We cannot simply *extend* the Java or Kotlin plugin to make it work. We need a plugin supporting *both* languages.

Unless – we extract some common APIs for syntax with bindings and refactor all plugins to contribute to the definition database.

LSP – the model

In an LSP-based editor:

- Adding a language support won’t affect other plugins and the code editor itself
- Adding an editor action requires an update on all plugins
- If an IDE action is not supported by a plugin, do nothing when invoked

It looks like subtyping-polymorphism in the expression problem!

LSP – the opposite

Is there an editor such that:

- Adding an editor action won’t affect other plugins and the code editor itself
- Adding a language support requires an update on all plugins
- If a language is not supported by a plugin, do nothing when any action is invoked on it

IntelliJ Platform & Visual Studio

—

Yes there is, but most IDEs of this kind also allow adding new languages support.

```
<lang.commenter language="Narc" implementationClass="org.ice1000.tt.editing.CxxLineCommenter"/>
<lang.refactoringSupport language="Narc" implementationClass="org.ice1000.tt.editing.InplaceRenameRefactoringSupportProvider"/>
<lang.braceMatcher language="Narc" implementationClass="org.ice1000.tt.editing.narc.NarcBraceMatcher"/>
<stubElementTypeHolder class="org.ice1000.tt.psi.narc.NarcTypes"/>
<lang.findUsagesProvider language="Narc" implementationClass="org.ice1000.tt.editing.narc.NarcFindUsagesProvider"/>
<lang.syntaxHighlighterFactory language="Narc" implementationClass="org.ice1000.tt.editing.narc.NarcHighlighterFactory"/>
<completion.contributor language="Narc" implementationClass="org.ice1000.tt.editing.narc.NarcCompletionContributor"/>
<lang.parserDefinition language="Narc" implementationClass="org.ice1000.tt.psi.narc.NarcGeneratedParserDefinition"/>
<colorSettingsPage implementation="org.ice1000.tt.editing.narc.NarcColorSettingsPage"/>
```

Ideal IDE design

- Turns out VSCode is taking a step backwards the technology progression
- Disregarding software-engineering, what are the pros & cons of both architectures?

Pros & Cons

- LSP-based: text buffer can be implemented better (lazy piece-table/rope)
- VS/IDEA: AST can be accessed more efficiently as it’s stored in the IDE
- Examples

Yet another option

- Self-contained editor

- Isabelle – jEdit
- Coq – CoqIde
- JetBrains MPS
- CovScript IDE/XStudio – CovScript/xlang

Thank you
and now let’s ask

—

jetbrains.com