# TypeScript is Surprisingly OK for Compilers

Aug 17, 2023

There are two main historical trends when choosing an implementation language for something compiler-shaped.

For more language-centric tasks, like a formal specification, or a toy hobby language, OCaml makes most sense. See, for example, plzoo or WebAssembly reference interpreter.

For something implementation-centric and production ready, C++ is often chosen: LLVM, clang, v8, HotSpot are all C++.

These days, Rust is a great new addition to the landscape. It is influenced most directly by ML and C++, combines their strengths, and even brings something new of its own to the table, like seamless, safe multithreading. Still, Rust leans heavily towards production readiness side of the spectrum. While some aspects of it, like a "just works" build system, help with prototyping as well, there's still extra complexity tax due to the necessity to model physical layout of data. The usual advice, when you start building a compiler in Rust, is to avoid pointers and use indexes. Indexes are great! In large codebase, they allow greater decoupling (side tables can stay local to relevant modules), improved performance (an index is `u32` and nudges you towards struct-of-arrays layouts), and more flexible computation strategies (indexes are easier to serialize or plug into incremental compilation framework). But they do make programming-in-the-small significantly more annoying, which is a deal-breaker for hobbyist tinkering.

But OCaml is crufty! Is there something better? Today, I realized that TypeScript might actually be OK? It is not really surprising, given how the language works, but it never occured to me to think about TypeScript as an ML equivalent before.

So, let's write a tiny-tiny typechecker in TS!

Of course, we start with deno. See *A Love Letter to Deno* for more details, but the TL;DR is that deno provides out-of-the-box experience for TypeScript. This is a pain point for OCaml, and something that Rust does better than either OCaml or C++. But deno does this better than Rust! It's just a single binary, it comes with linting and formatting, there's no compilation step, and there are built-in task runner and watch mode. A dream setup for quick PLT hacks!

And then there's TypeScript itself, with its sufficiently flexible, yet light-ceremony type system.

Let's start with defining an AST. As we are hacking, we won't bother with making it an IDE-friendly concrete syntax tree, or incremental-friendly "only store relative offsets" tree, and will just tag AST nodes with locations in file:

```
export interface Location {
  file: string;
  line: number;
  column: number;
}
```

Even here, we already see high-level nature of TypeScript — string is just a `string`, there's no thinking about `usize` vs `u32` as numbers are just `numbers`.

Usually, an expression is defined as a sum-type. As we want to tag each expression with a location, that representation would be slightly inconvenient for us, so we split things up a bit:

```
export interface Expr {
    location: Location;
    kind: ExprKind;
}

export type ExprKind = ExprBool | ExprInt | ... ;
```

One more thing — as we are going for something quick, we'll be storing inferred types directly in the AST nodes. Still, we want to keep raw and type-checked AST separate, so what we are going to do here is to parametrize the `Expr` over associated data it stores. A freshly parsed expression would use `void` as data, and the type checker will set it to `Type`. Here's what we get:

```
export interface Expr<T> {
  location: Location;
  data: T;
  kind: ExprKind<T>;
}

export type ExprKind<T> =
  | ExprBool<T>
  | ExprInt<T>
  | ExprBinary<T>
  | ExprControl<T>;
```

A definition of `ExprBinary` could look like this:

```ts
export interface ExprBinary<T> {
  op: BinaryOp;
  lhs: Expr<T>;
  rhs: Expr<T>;
}

export enum BinaryOp {
  Add, Sub, Mul, Div,
  Eq, Neq,
  Lt, Gt, Le, Ge,
}
```

Note how I don't introduce separate types for, e.g, `AddExpr` and `SubExpr` — all binary expressions have the same shape, so one type is enough!

But we need a tiny adjustment here. Our `Expr` kind is defined as a union type. To match a value of a union type a bit of runtime type information is needed. However, it's one of the core properties of TypeScript that it doesn't add any runtime behaviors. So, if we want to match on expression kinds (and we for sure want!), we need to give a helping hand to the compiler and include a bit of RTTI manually. That would be the `tag` field:

```ts
export interface ExprBinary<T> {
  tag: "binary";
  op: BinaryOp;
  lhs: Expr<T>;
  rhs: Expr<T>;
}
```

`tag: "binary"` means that the only possible runtime value for `tag` is the string `"binary"`.

Similarly to various binary expressions, boolean literal and int literal expressions have *almost* identical shape. Almost, because the payload (`boolean` or `number`) is different. TypeScript allows us to neatly abstract this over:

```ts
export type ExprBool<T> = ExprLiteral<T, boolean, "bool">;
export type ExprInt<T> = ExprLiteral<T, number, "int">;

export interface ExprLiteral<T, V, Tag> {
  tag: Tag;
  value: V;
}
```

Finally, for control-flow expressions we only add `if` for now:

```ts
export type ExprControl<T> = ExprIf<T>;

export interface ExprIf<T> {
  tag: "if";
```

```
  cond: Expr<T>;
  then_branch: Expr<T>;
  else_branch: Expr<T>;
}
```

This concludes the definition of the ast! Let's move on to the type inference! Start with types:

```
type Type = TypeBool | TypeInt;

interface TypeBool {
  tag: "Bool";
}
const TypeBool: TypeBool = { tag: "Bool" };

interface TypeInt {
  tag: "Int";
}
const TypeInt: TypeInt = { tag: "Int" };
```

Our types are really simple, we could have gone with

```
type Type = "Int" | "Bool",
```

but lets do this a bit more enterprisy! We define separate types for integer and boolean types. As these types are singletons, we also provide canonical definitions. And here is another TypeScript-ism. Because TypeScript fully erases types, everything related to types lives in a separate namespace. So you can have a type and a value sharing the same name. Which is exactly what we use to define the singletons!

Finally, we can take advantage of our associated-data parametrized expression and writhe the signature of

```
function infer_types(expr: ast.Expr<void>): ast.Expr<Type>
```

As it says on the tin, `inter_types` fills in `Type` information into the void! Let's fill in the details!

```
function infer_types(expr: ast.Expr<void>): ast.Expr<Type> {
  switch (expr.kind.tag) {
    cas
  }
}
```

If at this point we hit Enter, the editor completes:

```
function infer_types(expr: ast.Expr<void>): ast.Expr<Type> {
  switch (expr.kind.tag) {
```

```
    case "bool":
    case "int":
    case "binary":
    case "if":
  }
}
```

There's one problem though. What we really want to write here is something like

```
const inferred_type = switch(..),
```

but in TypeScript `switch` is a statement, not an expression. So let's define a generic visitor!

```
export type Visitor<T, R> = {
  bool(kind: ExprBool<T>): R;
  int(kind: ExprInt<T>): R;
  binary(kind: ExprBinary<T>): R;
  if(kind: ExprIf<T>): R;
};

export function visit<T, R>(
  expr: Expr<T>,
  v: Visitor<T, R>,
): R {
  switch (expr.kind.tag) {
    case "bool": return v.bool(expr.kind);
    case "int": return v.int(expr.kind);
    case "binary": return v.binary(expr.kind);
    case "if": return v.if(expr.kind);
  }
}
```

Armed with the `visit`, we can write ergonomically match over the expression:

```
function infer_types(expr: ast.Expr<void>): ast.Expr<Type> {
  const ty = visit(expr, {
    bool: () => TypeBool,
    int: () => TypeInt,
    binary: (kind: ast.ExprBinary<void>) => result_type(kind.op),
    if: (kind: ast.ExprIf<void>) {
      ...
    },
  });
  ...
}

function result_type(op: ast.BinaryOp): Type {
  switch (op) { // A tad verbose, but auto-completed!
```

```
    case ast.BinaryOp.Add: case ast.BinaryOp.Sub:
    case ast.BinaryOp.Mul: case ast.BinaryOp.Div:
      return TypeInt

    case ast.BinaryOp.Eq: case ast.BinaryOp.Neq:
      return TypeBool

    case ast.BinaryOp.Lt: case ast.BinaryOp.Gt:
    case ast.BinaryOp.Le: case ast.BinaryOp.Ge:
      return TypeBool
  }
}
```

Before we go further, let's generalize this visiting pattern a bit! Recall that our expressions are parametrized by the type of associated data, and type-checker-shaped transformations are essentially an

```
Expr<U> -> Expr<V>
```

transformation.

Let's make this generic!

```
export function transform<U, V>(expr: Expr<U>, v: Visitor<V, V>): Expr<V>
```

Transform maps an expression carrying `T` into an expression carrying `V` by applying an `f` visitor. Importantly, it's `Visitor<V, V>`, rather than a `Visitor<U, V>`. This is counter-intuitive, but correct — we run transformation bottom up, transforming the leaves first. So, when the time comes to visit an interior node, all subexpression will have been transformed!

The body of `transform` is wordy, but regular, rectangular, and auto-comptletes itself:

```
export function transform<U, V>(expr: Expr<U>, v: Visitor<V, V>): Expr<V>
  switch (expr.kind.tag) {
    case "bool":
      return {
        location: expr.location,
        data: v.bool(expr.kind),
        kind: expr.kind, 1
      };
    case "int":
      return {
        location: expr.location,
        data: v.int(expr.kind),
        kind: expr.kind,
      };
    case "binary": {
      const kind: ExprBinary<V> = { 2
```

```
        tag: "binary",
        op: expr.kind.op,
        lhs: transform(expr.kind.lhs, v),
        rhs: transform(expr.kind.rhs, v),
      };
      return {
        location: expr.location,
        data: v.binary(kind),  2
        kind: kind,
      };
    }
    case "if": {
      const kind: ExprIf<V> = {
        tag: "if",
        cond: transform(expr.kind.cond, v),
        then_branch: transform(expr.kind.then_branch, v),
        else_branch: transform(expr.kind.else_branch, v),
      };
      return {
        location: expr.location,
        data: v.if(kind),
        kind: kind,
      };
    }
  }
}
```

1 Note how here `expr.kind` is both `Expr<U>` and `Expr<V>` — literals don't depend on this type parameter, and TypeScript is smart enough to figure this out without us manually re-assembling the same value with a different type.

2 This is where that magic with `Visitor<V, V>` happens.

The code is pretty regular here though! So at this point we might actually recall that TypeScript is a dynamically-typed language, and write a generic traversal using `Object.keys`, *while keeping the static function signature in-place*. I don't think we need to do it here, but there's comfort in knowing that it's possible!

*Now* implementing type inference should be a breeze! We need some way to emit type errors though. With TypeScript, it would be trivial to accumulate errors into an array as a side-effect, but let's actually represent type errors as instances of a specific type, `TypeError` (pun intended):

```
type Type = TypeBool | TypeInt | TypeError;

interface TypeError {
```

```
  tag: "Error";
  location: ast.Location;
  message: string;
}
```

To check ifs and binary expressions, we would also need an utility for comparing types:

```
function type_equal(lhs: Type, rhs: Type): boolean {
  if (lhs.tag == "Error" || rhs.tag == "Error") return true;
  return lhs.tag == rhs.tag;
}
```

We make the `Error` type equal to any other type to prevent cascading failures. With all that machinery in place, our type checker is finally:

```
function infer_types(expr: ast.Expr<void>): ast.Expr<Type> {
  return ast.transform(expr, {
    bool: (): Type => TypeBool,
    int: (): Type => TypeInt,

    binary: (kind: ast.ExprBinary<Type>, location: ast.Location): Type =>
      if (!type_equal(kind.lhs.data, kind.rhs.data)) {
        return {
          tag: "Error",
          location,
          message: "binary expression operands have different types",
        };
      }
      return result_type(kind.op);
    },

    if: (kind: ast.ExprIf<Type>, location: ast.Location): Type => {
      if (!type_equal(kind.cond.data, TypeBool)) {
        return {
          tag: "Error",
          location,
          message: "if condition is not a boolean",
        };
      }
      if (!type_equal(kind.then_branch.data, kind.else_branch.data)) {
        return {
          tag: "Error",
          location,
          message: "if branches have different types",
        };
      }
      return kind.then_branch.data;
    },
  });
```

```
}

function result_type(op: ast.BinaryOp): Type {
    ...
}
```

Astute reader will notice that our visitor functions now take an extra `ast.Location` argument. TypeScript allows using this argument only in cases where it is needed, cutting down verbosity.

And that's all for today! The end result is pretty neat and concise. It took some typing to get there, but TypeScript autocompletion really helps with that! What's more important, there was very little fighting with the language, and the result feels quite natural and directly corresponds to the shape of the problem.

I am not entirely sure in the conclusion just yet, but I think I'll be using TypeScript as my tool of choice for various small language hacks. It is surprisingly productive due to the confluence of three aspects:

- deno is a perfect scripting runtime! Small, hermetic, powerful, and optimized for effective development workflows.
- TypeScript tooling is great — the IDE is helpful and productive (and deno makes sure that it also requires zero configuration)
- The language is powerful both at runtime and at compile time. You can get pretty fancy with types, but you can also just escape to dynamic world if you need some very high-order code.

---

Just kidding, here's one more cute thing. Let's say that we want to have lots of syntactic sugar, and also want type-safe desugaring. We could tweak our setup a bit for that: instead of `Expr` and `ExprKind` being parametrized over associated data, we circularly parametrize `Expr` by the whole `ExprKind` and vice verse:

```
interface Expr<K> {
  location: Location,
  kind: K,
}

interface ExprBinary<E> {
  op: BinaryOp,
  lhs: E,
  rhs: E,
}
```

This allows expressing desugaring in a type-safe manner!

```ts
// Fundamental, primitive expressions.
type ExprKindCore<E> =
    ExprInt<E> | ExprBinary<E> | ExprIf<E>

// Expressions which are either themselves primitive,
// or can be desugared to primitives.
type ExprKindSugar<E> = ExprKindCore<E>
    | ExprCond<E> | ExprUnless<E>

type ExprCore = Expr<ExprKindCore<ExprCore>>;
type ExprSugar = Expr<ExprKindSugar<ExprSugar>>;

// Desugaring works by reducing the set of expression kinds.
function desugar(expr: ExprSugar): ExprCore

// A desugaring steps takes a (potentially sugar) expression,
// whose subexpression are already desugared,
// and produces an equivalent core expression.
function desugar_one(
    expr: ExprKindSugar<ExprCore>,
): ExprKindCore<ExprCore>
```