Refining Types using Type Guards in TypeScript

Ivo Gabe de Wolff
Department of Information and Computing Sciences,
Utrecht University, The Netherlands
i.g.dewolff@uu.nl

Jurriaan Hage
Department of Information and Computing Sciences,
Utrecht University, The Netherlands
J.Hage@uu.nl

Abstract

We discuss two adaptations of the implementation of type guards and narrowing in the TypeScript compiler. The first is an improvement on the original syntax-directed implementation, and has now replaced the original one in the TypeScript compiler. It is specifically suited for the scenario in which an IDE requests the type of a particular variable in the program. The second implementation is defined as a whole program analysis, and is therefore able to compute more precise narrowed types, but at the price of a higher run-time cost.

Categories and Subject Descriptors F.3.3 [*Semantics of Programming Languages*]: Progam analysis

Keywords type systems, type error diagnosis, TypeScript

1. Introduction

JavaScript is an untyped dynamic language, originally created to make websites interactive. TypeScript is a language based on JavaScript, created by Microsoft, that compiles to JavaScript. It is a superset of JavaScript, adding type annotations that are checked at compile time. TypeScript is used on a number of large projects, including Microsoft's Azure Portal (over one 1 mln lines) and Shumway (at 170.000 lines) (Turner 2014, 2015).

JavaScript is a dynamic language, so not all constructs can be easily checked by a classic type checker. One such case is when a variable has multiple types. For instance, it is possible to write the following:

```
  function f(x: string | string[]): string {
    if (typeof x === "string") {   // string    }
    else { // string[]  }
  }
```

in which the variable x may refer to a string or to an array of strings. The type of such a parameter can be examined at runtime; we call a condition that does so a *type guard*.

A classic type checker may not exploit the veracity of the condition in its type analysis. That means that you still have to cast the variable after a type guard, although clearly the then-part can only be reached if x has type `string`, while in the else-part x must have type `string[]`.

The TypeScript compiler does exploit type guards. This means that a type guard, which was originally only a runtime check, is now used at compile time too. Setting a type to a more specific type than its declared type is called *narrowing* or *refining*. The syntax-directed implementation in the original TypeScript compiler has some shortcomings. Consider the following example program:

```
function f(x: string | string[]) {
        if (typeof x === "string") {
                return;
        }
        x;
}
```

The type of x on the last line as established by the original compiler implementation is `string | string[]`, whereas `string[]` is desired. The reason for this is that the original implementation was based on the abstract syntax tree, not on the actual control-flow graph.

We have constructed two implementations to improve the existing implementation, each geared to a particular scenario. In the first scenario, the compiler is used from inside an editor/IDE, often after a small change to the source file. In that case, we can expect only few types to be changed whenever something happens. The first implementation we discuss is suitable for this scenario, and fits well into the architecture of the TypeScript compiler. For example, it can be used to give type information in editors: for a single position of a variable, it is able to show the type quickly without analyzing the whole program.

We have implemented this approach in a fork of TypeScript. Anders Hejlsberg, one of the core contributors of TypeScript, has made an (performance) improved implementation based on ours, which is now part of the TypeScript compiler (Rosenwasser 2016). The code for the control flow based implementation can be found in (de Wolff 2016b) and the implementation of Anders Hejlsberg in (Hejlsberg 2016a). During our research, he has also added checking for `null` and `undefined`, which also benefits a lot from the control flow based type analysis. This is discussed in section 7 along with other use cases.

As a second contribution, we have made a second standalone implementation based on monotone frameworks (Nielson et al. 1999) which can achieve higher precision, but this is a whole program analysis. This analysis can be used when the program is compiled to JavaScript, after which the incremental implementation can take over again. The monotone framework implementation can be found in (de Wolff 2016a).

The previous examples might look theoretical and not realistic. Section 7 gives some practical use-cases of the feature, including the narrowing of algebraic data types and the checking of `null` and `undefined` values.

In Section 2 we introduce important parts of TypeScript, discussing narrowing and type guards in Section 3. Section 4 then discusses the control-flow based local approach to narrowing, while Section 5 describes the monotone framework implementation. In Section 6 we provide some preliminary performance benchmarks

PEPM'17, January 16–17, 2017, Paris, France
© 2017 ACM. 978-1-4503-4721-1/17/01...$15.00
http://dx.doi.org/10.1145/3018882.3018887

for both implementations. Section 7 discusses some use cases in which our analyses can be employed. Finally, Section 8 discusses related work, and Section 9 provides some possibilities for future work and concludes.

This paper describes the state of TypeScript 1.8. Changes made to the language since then are documented at `https://github.com/Microsoft/TypeScript/wiki/Roadmap`.

2. The Types of TypeScript

In this section we shortly introduce TypeScript, in particular the aspects of its type system relevant to this paper. We have made some choices in our analysis in relation to the types, which we document here as well.

TypeScript supports a large number of primitive types: `never` denotes the bottom type, `any` the top of the type lattice. In addition we have `void`, `undefined`, `null`, `number`, `boolean`, and `string`.

The keyword `let` can be used to declare a variable, so that `let x : T` means that a variable `x` is declared with `T` as its type upper-bound. Variables can also be declared with `const` and `var`, but they all mean the same thing when considering only the types. A small example follows:

```
let x: number;
x = 0; // Ok
x = "foo"; // Error
```

For object types we list a number of properties with their type. A special aspect is that properties ending in `?` are optional; this means objects do not have to have such a property, but if they do, they should have the correct type. An example of such an object type is:

```
interface Foo {
        // Properties
        x: number;
        y: Foo;
        // Optional property
        z?: boolean;
}
let a: Foo = {
        x: 42,
        y: undefined
};
let b: Foo = {
        x: NaN,
        y: a,
        z: true
};
```

An object type can have call signatures, as functions are objects too. An overloaded function has multiple signatures.

```
interface Bar {
        // Call signature
        (x: number): number;
        // Optional property
        foo?: boolean;
        // Optional property, numeric name
        0?: boolean;
}
let c: Bar = (x: number) => x * 2;
// Optional properties can be set later.
c.foo = true;
c[0] = true;
```

Primitive types have properties like `toString` and `toExponentional` making them assignable to object types:

```
interface HasToString {
        toString: () => string;
}
let g: HasToString;
g = 42;   // Numbers have a toString property
g = true; // Booleans too
```

Indexers are wildcard properties; they introduce a set of properties whose identifier matches the type specified in the indexer:

```
interface StringIndexer {
        [x: string]: boolean;
}
let d: StringIndexer = {};
d['foo'] = true;
d['bar'] = false;

interface NumberIndexer {
        [x: number]: boolean;
}
```

Object literals are a notation to create objects. The notation consists of an opening curly brace, a set of properties separated by commas, and a closing curly brace. As of TypeScript 1.6, an object literal may not contain unknown (as in, not defined in the specified type) properties, which is why the definition of `c` is illegal. However, we can assign such a value to `a`, and then copy it to `b`.

```
interface Point {
        x: number;
        y: number;
}
interface Point3D {
        x: number;
        y: number;
        z: number;
}
let a: Point3D = { x: 1, y: 1, z: 1 }; // Ok
let b: Point = a;                      // Ok
let c: Point = { x: 1, y: 1, z: 1 };   // Error
```

These extra, unknown properties are called *excess properties* in the language specification (Microsoft 2016c).

Enum types are very much like their cousins in other languages. Of note is that for enums the type system behaves nominally, values of one enum type are not assignable to variables of another enum type.

TypeScript also supports intersection and union types: a union type, denoted as `A | B`, is a type that is assignable to `A` or `B`. The set of assignable values of a union type is the union of the sets of `A` and `B`, hence the name. The type `never` is defined as an empty union type. An intersection type, `A & B`, is assignable to both `A` and `B`. From these definitions it follows that `A & (B | C)` is equivalent to `(A & B) | (A & C)`. The set of assignable values of an intersection type is the intersection of the sets of the parts.

A *function type* consists of a list of type parameters, a list of arguments and a return type. Parameters have a name and a type. The last parameter may be a rest parameter, which is used to define functions with an arbitrary number of arguments (like `Math.max`).

```
let add: (x: number, y: number) => number;
let id: <U>(x: U) => U;  // Polymorphic

let a: { (x: number) => number
        , property: string };
let b: ((x: number) => number)
        & { property: string };
```

In the definition of `a` and `b`, the types are in fact equal, but written in a different fashion.

As of TypeScript 1.8, users can write a type to which only one string is assignable. Combined with union types, this can be used to model magic strings. Magic strings, common in JavaScript, are an alternative to enumerations, as in

```
type Day = "sat" | "sun" | "mon" | "tue" | "wed"
         | "thu" | "fri";
let day: Day;
day = "sat"; // Ok
day = "may"; // Error
```

Literal types will be extended to number, enum and boolean types (Hejlsberg 2016b). In our analysis, we support string, number and boolean literal types.

By default, every type is nullable, which means that `undefined` and `null` can be assigned to variables of every type. With a new option of TypeScript 2.0, called `strictNullChecks`, `undefined` and `null` are two new types and no other types can contain these values. If you want to declare a variable that could be a string or `null`, you must annotate it with a union type `string | null`. We will do the analysis in this mode.

3. Type Guards and Narrowing

A *type guard* is a condition that checks whether a value is of a certain type. If that value references a variable, `this`, or a property of one of those, the compiler can change the type of the reference in the scope of the type guard; this is called *narrowing*. For example,

```
let x: string | number | boolean;
if (typeof x !== "string") {
        if (typeof x !== "number") {
                // x: boolean
        }
}
// Or we can write:
if (typeof x !== "string" &&
    typeof x !== "number") {
        // x: boolean
}
```

TypeScript has the following type guards built in:

- `typeof` - narrows to primitive types
- `instanceof` - narrows a class type to a subclass
- equality checks - intersects the types of both operands
- truthiness checks - narrows to all truthy or falsy types. A type is truthy if it is `true` after conversion to a boolean, falsy if it becomes `false`.

Additional examples of these type guards are given in Figure 1.

In the monotone framework implementation, we have added support for comparison type guards for <, >, <= and >=, in case our types are (unions of) literal types. For `a < b`, we take the largest value of `b`, and remove of subtypes of `a` that are larger than or equal to that value. Similarly, the type of `b` is narrowed. This is not always precise: the type guard for `a < a` should preferably yield `never`, but in our approach it does not.

A complication for our analysis is that TypeScript allows the definition of user-defined type guards, as illustrated in Figure 2, in which the syntax `is Cat` indicates that the return value of the function is refined to the `Cat` subtype.

4. The Local Control-Flow Based Analysis

The original implementation in the TypeScript compiler was syntax directed, following the shape of the AST. This means for example, that a node (which should be a type guard of some kind) in the AST can only narrow the type of its children. However, in some cases this is not intuitive:

```
let x: number | Array<boolean> | "foo" = ...;
if (x instanceof Array) {
        // x: Array<boolean>
} else {
        // x: number | "foo";
}
if (x === "foo") {
        // x: "foo"
} else {
        // x: number | Array<boolean>;
}
let y: number | string | boolean = ...;
if (x === y) {
        // x: number | "foo"  y: number | "foo"
} else {
        // x: number | Array<boolean> | "foo"
        // y: number | string | boolean
}
let z: number | undefined = ...;
if (z) {
        // z: number
} else {
        // z: 0 | undefined
}
```

Figure 1. Type guard examples

```
interface Animal {
        name: string;
}
interface Cat extends Animal {
        meow(): void;
}

// User defined type guard
function isCat(animal: Animal): animal is Cat {
        return animal.name === "Kitty";
}
let x: Animal;
if (isCat(x)) {
        // x: Cat
        x.meow();
}
```

Figure 2. A user-defined type guard for cats

```
let x: string | number = ...;
if (typeof x === "string") {
        // x: string
}
if (typeof x === "string") {
        // x: string | number
        x = "a";
        // x: string | number
}
```

In the latter example, the assignment blocks the refinement for the whole block, and no advantage of the type guard can be had. This is something that is hard to explain to a programmer, and one situation we would like to amend. The syntax-directedness also makes the original implementation blind to assignments made during a function call, so sometimes types are narrowed when they should not be as the following example shows.

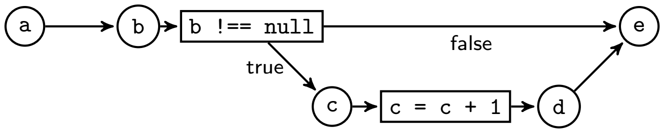

Figure 3. A simple control flow graph

```
let x: string | number = ...;
function setX() {
        x = 42;
}
if (typeof x === "string") {
        setX();
        // number at runtime, compiler says string
}
```

The designers of TypeScript decided to accept this unsoundness; one of the non-goals says: *"Apply a sound or 'provably correct' type system. Instead, strike a balance between correctness and productivity"* (Microsoft 2016b).

In the remainder of this section we discuss our improved, incremental control-flow based implementation that has found its way into the current version of the TypeScript compiler (thanks to Anders Hejlsberg). Our implementation shares one property of the original implementation: it is given the location of a variable, and will only analyze those parts of the program that contribute to that type. The idea is that such an analysis is much faster than a whole program analysis so that it can be used in an IDE. To compile a whole project from scratch, this process is invoked repeatedly to compute all types in the project, cleverly keeping parts of the results of these separate invocations. The approach discussed in Section 5 analyzes the complete project in a single analysis.

The control flow graph can be implemented by giving each node of the AST an adjacency list containing the prior nodes in the control flow. Using this representation, the previous occurrences of a variable can easily be found. However, not all nodes of the AST have to be in the graph. In our implementation, only identifiers, type guards, assignments and nodes that can change the control flow (such as an if- or for-statement) are part of the graph. For the following code fragment

```
a();
if (b !== null) {
        c = c + 1;
        d();
}
e();
```

our implementation generates the control-flow graph in Figure 3, where a circle represents an identifier and a rectangle an assignment or type guard.

It is common that large pieces of the graph are straight-line code sections, with no branches nor assignments. For instance, a long expression can contain a lot of variables. Anders Hejlsberg adapted our implementation in which the graph contains only one node for these straight lines. For our example, the graph in Figure 4 results.

Our initial implementation was 40 times slower than the original compiler with the syntax-directed implementation. We reduced this number to 20 by replacing recursion with a manual stack, tracked in an array (de Wolff 2016b). The implementation of Anders Hejlsberg, which has a smaller control flow graph, was approximately as fast as the syntax-directed implementation. His implementation does use recursion, but includes explicit tail recursion for nodes of the control flow graph that have only one antecedent (Hejlsberg 2016a).

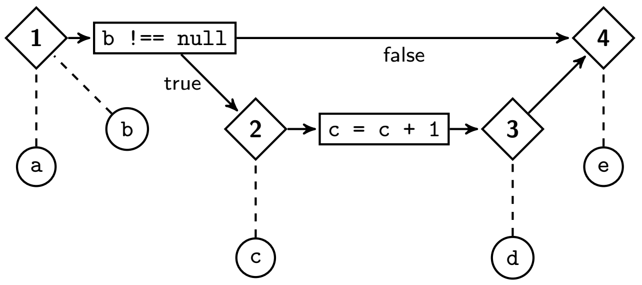

Figure 4. The modified control graph

4.1 Narrowing after Assignments

The type of a variable will be narrowed after an assignment. Consider the assignment `x = e`. If the type of `x` were directly narrowed to the type of the assigned value, you can get unexpected behavior, as demonstrated by these examples.

```
let x: { a?: boolean };
x = {};
x.a = true;

class Base { a; }
class Derived extends Base { b; }
let y: Base[];
y = [new Derived()];
y.push(new Base());
```

If `x` were narrowed to $\{\}$, the empty object type, the assignment to `x.a` would be invalid. If `y` were narrowed to `Derived[]`, it would be an error to push an object of type `Base` to the array. To prevent these issues, only union types will be narrowed: the resulting type is the union of the parts of the original union type to which the type of `e` is assignable. In the following example, `x` is narrowed to `Base`, since `Derived` is assignable to `Base` but not to `string`.

```
let x: Base | string;
x = new Derived();
```

In the next example, `x` would be narrowed to the empty type with this rule. `Derived | string` is assignable to neither `Base`, `string` nor `number`, so this rule does not work as expected. This can be fixed by applying the previous rule to each part of an assigned union type. For this example that would mean that the filtering on `Derived` results in `Base` and the filtering on `string` results in `string`, combined this yields `Base | string`.

```
let x: Base | string | number;
let e: Derived | string;
x = e;
```

4.2 Branches

If code has a lot of branches, the graph can have an exponential number of paths between two flow nodes. Consider a file with n consecutive if-statements. Then 2^n paths between the top and bottom of the file exist. When analyzing all paths separately, this would cost $\Omega(2^n)$, which will be too slow for large code files. We can improve the performance by storing the type of a variable before each branching node, such as an if-statement, as a form of dynamic programming. In the example, this means that for each if-statement only two nodes should be evaluated, in total $2n$.

4.3 Iterations

Iterations can complicate the analysis. Given that the body of a loop can be evaluated multiple times, the types in the body depend on themselves. The following example shows such a case.

```
interface A { a; x: B; }
interface B { b; x: C; }
interface C { c; x: A; }
let x: A | B | C = {a: 0; x: ...};
while (true) {
        x = x.x;
        x;
}
```

On the last line of the `while` loop, the type of `x` should be `A | B | C`. Before the loop, `x` is known to be of type `A`. In the first iteration, the type of `x` will have type `B`, in the next iteration this becomes `C`, and then back to `A`. This can be implemented with a least fixed point iteration, starting with `A`, converging to the type of `x` for which `x = x.x`. This iteration would analyze along the back edge multiple times until the type stabilizes.

Given that assignments only filter a (finite) union type, this process will terminate if no type guards are present. However, it is possible to get an infinite chain of types with type guards. When a user-defined type guard has both the variable that should be narrowed as the previous value as arguments, this can lead to an infinite chain. Imagine that a type guard `isOneHigher` exists that checks whether the first argument is one higher than the second argument. The following code would give an infinite chain of types; the type of `x` is $a_i = 0|1|...|i$ when the loop has been analyzed i times.

```
let x: number;
while (true) {
        const y = x;
        x = ...;
        if (!isOneHigher(x, y)) return;
        // x: 0 | 1 | 2 | ...
}
```

It is currently not possible to write such a type guard. However, an infinite chain of types can also be constructed with objects with generics:

```
interface Wrap<T> { value: T; }
function wrapContains<U, V extends U>
           (w: Wrap<U>, value: V): w is Wrap<V> {
        ...
}

let x: Wrap<any> = ...;
if (!wrapContains(x, "")) return;
// x: Wrap<string>
while (1) {
        const y = x;
        x = ...;
        if (!wrapContains(x, y)) return;
}
```

After analyzing the path entering the loop, the type of `x` after the last if-statement would be `Wrap<Wrap<string>>`. When analyzing the back edge repeatedly, arbitrarily deeply nested objects will be found. This can be fixed by limiting the number of times that the back edge is analyzed, and falling back to the initial type of the variable (`Wrap<any>` in this case) when that limit is exceeded. This is not always accurate, but it is sound and can improve the performance. A simpler solution would be to fall back to the declared type if the type of a variable depends on the type on the same location. TypeScript already does something similar for other recursive constructs. For instance, the return type of a function without a return type annotation that calls itself recursively cannot be inferred and falls back to `any`.

An exception to this rule is the case where the type of a variable depends directly on the type at that location, or more formally, there is a path from the location to itself, with no assignments to that variable. This path can contain type guards. The type for this path can only be a subtype of the type at that location. Thus, we may ignore this path in the analysis.

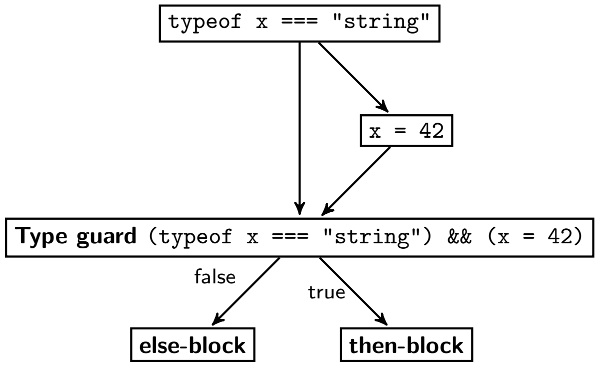

Figure 5. A control-flow graph for assignments in conditions

4.4 Assignments in Type Guards

The control flow based implementation can have issues when a type guard contains an assignment. Consider the following code:

```
let x: string | number | boolean = ...;
if ((typeof x === "string") && (x = 42)) {
        x;
}
```

Figure 5 shows the graph that is generated if the condition of the if-statement is placed directly as one node in the graph. When analyzing the type of `x` in the then-block, the path from the assignment `x = 42` contains the type guard `(typeof x === "string") && (x = 42)`. Applying it narrows the type of `x` based on `(typeof x === "string")` and `(x = 42)`. The first of those is a type guard, which narrows `number` to `never`. The second is not a type guard, and does not narrow. However, at runtime the value of `x` can only be a number there.

Expanding these logical expressions does not fix all issues. An operand of a type guard may modify the value of the variable. If that operand is evaluated after the getting the value of the variable, narrowing should be blocked. This may happen when user defined type guards are employed:

```
let x: string | number = undefined as any;
function isString(x: any, y: any): x is string {
        return typeof x === "string";
}

if (isString(x, x = 4)) {
        x;
}
```

To fix these issues, we must not treat a logical binary expression (`&&` or `||`) as a type guard, but we must expand it as can be seen in Figure 6. Note that the same scenario can occur in an `instanceof` check, for instance `x instanceof (x = 1, Object)`.

The syntax of a `typeof` type guard is too restrictive to deal with these cases. The value of `typeof` may only be compared with a string literal, so `typeof x === "str" + "ing"` does not narrow. Our monotone framework implementation does allow more freedom here. For instance, a `typeof` check may be compared to any value. Thus, `typeof x === (x = 1, "string")` is allowed. The comma operator evaluates the first and the second operand and returns the second operand. The assignment to `x` is executed after retrieving the value of `x` for the `typeof`, so the type guard works on the old value of `x`. Narrowing should be blocked in this case.

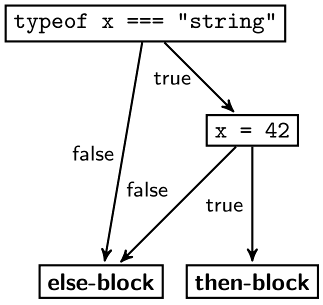

Figure 6. A fixed control-flow graph

4.5 Algorithm

The compiler performs a bind step in a single pass over the AST setting the scopes of all variables, and reporting errors for unreachable code. We have adapted this traversal by replacing the reachability checks by the creation of the control flow graph.

To avoid confusion, a node from the AST is called a node and a node from the control flow graph a flow node. The function `union` takes the union of several types, `typeNever` represents the type `never`, `initialType` the initial type of the analyzed variable and `node` the location of the variable. The function `narrow` narrows the type of a variable based on a type guard and that function should check whether the type guard contains an assignment.

As discussed before, iterations can be handled in two ways. When analyzing a loop, the body of the loop can cause recursion. A simple implementation falls back to the initial type in these cases; it can be implemented based on the pseudo-code in Figure 7.

An implementation using a least fixed point iteration is more accurate for loops. The pseudo code in Figure 8 demonstrates such an implementation, which limits the number of iterations to guarantee termination. The code uses `typeUsedRecursively` and `typesCache`, which are two dictionaries that store for each node a type and a boolean whether the type was used recursively. The latter is used to prevent starting a least fixed point iteration in cases where this iteration is not needed. This means that this implementation is only slower if a least fixed point iteration could give more accurate types. The constant `maxIterations` is the maximum number of times that the least fixed point iteration may be executed before falling back to the initial type.

4.6 Limitations

The implementation still has a few limitations. It is not interprocedural, which means that assignments in other functions are not handled. Moreover, it will also analyze code that could be known to be unreachable.

5. Monotone Framework Implementation

Our second implementation is based on the concept of monotone frameworks (Nielson et al. 1999). This implementation gives more accurate type information. Consider the following example: `let x = 0; x = 1`.

The standard TypeScript compiler will infer the type of `x` based on its initializer. If the compiler would treat that as `0`, the second line would be invalid. If it were treated as `number`, accuracy is lost. The creators of TypeScript have chosen the second option, since the first would require the programmer to write a lot more type annotations and it would be a breaking change from the previous versions. With a monotone framework, we can infer the type to be `0` first, and after the second line we can set it to `1`. If a variable has a type annotation, we can use it as an upper bound: we take the meet of the declared and assigned types. If that is not equal to the assigned type, the assignment is invalid and an error should be shown.

```
if node is already being analyzed
        return initialType
end

path := []
cache := {}

return typeAt(node.flowNode)

function typeAt(flowNode)
  if cache contains flowNode
    return cache[flowNode]
  end
  if flowNode is assignment to this variable
    return narrowByAssignment(initialType,
                              flowNode)
  end
  if path.contains(flowNode)
    return typeNever
  end
  path.push(flowNode)
  type := typeNever
  for antecedent of flowNode.antecedents
    type := union(type, typeAt(antecedent))
    if type equals initialType
      break
    end
  end
  if flowNode is type guard for this variable
    type := narrow(type, flowNode)
  end
  path.pop()
  cache[flowNode] := type
  return type
end
```

Figure 7. The algorithm without fixed point iteration

Another advantage of this implementation is that it can ignore code that is unreachable based on type information. This can be useful for a function that is polymorphic, in combination with an interprocedural analysis.

A monotone framework consists of a lattice L, a set of monotone transfer functions $\mathcal{F}$, the program flow F, a set of extremal program points E, an extremal value $\iota \in L$ and a mapping from program points to transfer functions $\lambda l.f_l$. An element of the lattice L is the state of the analysis at a certain program point. The bottom element $\bot \in L$ means that the program point is unreachable. Any other element contains the types of all variables at that location. The extremal program points are the first nodes of all files. Entry points of functions are added later in the inter-procedural analysis (Nielson et al. 1999; Fritz and Hage 2014).

To reduce complexity, we will not support the full TypeScript language. We will restrict the JavaScript features to the ECMAScript 5 specification and thus not support features like classes and destructuring from later ECMAScript versions. The implementation will only resolve types and will not check the types.

In the next sections we will discuss the type lattice and the transfer functions.

```
if node in typesCache
  typeUsedRecursively[node] := true
  return typesCache[node]
end
visited := []
cache := {}
type := typeAt(node.flowNode)
if !typeUsedRecursively[node]
  return type
end
i := 0
do
  i := i + 1
  if (i > maxIterations) return initialType
  previous := type
  cache := {}
  type := typeAt(node.flowNode)
while previous not equals type

function typeAt(flowNode)
  if cache contains flowNode
    return cache[flowNode]
  end
  if flowNode is assignment to this variable
    return narrowByAssignment(initialType,
                              flowNode)
  end
  if path.contains(flowNode)
    return typeNever
  end
  path.push(flowNode)
  type := typeNever
  if flowNode equals node.flowNode
    typesCache[node] := type
    typeUsedRecursively[node] := false
  end
  for antecedent of flowNode.antecedents
    type := union(type, typeAt(antecedent))
    if flowNode equals node.flowNode
      typesCache[node] := type
    end
    if type equals initialType
      break
    end
  end
  if flowNode is type guard for this variable
    type := narrow(type, flowNode)
  end
  path.pop()
  cache[flowNode] := type
  return type
end
```

Figure 8. The algorithm with fixed point iteration

5.1 Type Lattice

The type lattice is defined by defining the subtype relation. For this we employ the subtype definition from the 1.8 language specification with some modifications, as explained below.

5.1.1 Subtype Relation

The subtype relation is defined in the language specification version 1.8 as follows (Microsoft 2016c, as reproduced from section 3.11.3):

S is a subtype of a type T, and T is a supertype of S, if S has no excess properties with respect to T (3.11.5) and one of the following is true:

i. S and T are identical types.

ii. T is the `any` type.

iii. S is the `undefined` type.

iv. S is the `null` type and T is not the `undefined` type.

v. S is an enum type and T is the primitive type `number`.

vi. S is a string literal type and T is the primitive type `string`.

vii. S is a union type and each constituent type of S is a subtype of T.

viii. S is an intersection type and at least one constituent type of S is a subtype of T.

ix. T is a union type and S is a subtype of at least one constituent type of T.

x. T is an intersection type and S is a subtype of each constituent type of T.

xi. S is a type parameter and the constraint of S is a subtype of T.

xii. S is an object type, an intersection type, an enum type, or the `number`, `boolean`, or `string` primitive type, T is an object type, and for each member M in T, one of the following is true:

- M is a property and S has an apparent property N where
  - M and N have the same name,
  - the type of N is a subtype of that of M,
  - if M is a required property, N is also a required property, and
  - M and N are both public, M and N are both private and originate in the same declaration, M and N are both protected and originate in the same declaration, or M is protected and N is declared in a class derived from the class in which M is declared.
- M is a non-specialized call or construct signature and S has an apparent call or construct signature N where, when M and N are instantiated using type `any` as the type argument for all type parameters declared by M and N (if any),
  - the signatures are of the same kind (call or construct),
  - M has a rest parameter or the number of non-optional parameters in N is less than or equal to the total number of parameters in M,
  - for parameter positions that are present in both signatures, each parameter type in N is a subtype or supertype of the corresponding parameter type in M, and
  - the result type of M is `void`, or the result type of N is a subtype of that of M.
- M is a string index signature of type U, and U is the `any` type or S has an apparent string index signature of a type that is a subtype of U.

- `M` is a numeric index signature of type `U`, and `U` is the `any` type or `S` has an apparent string or numeric index signature of a type that is a subtype of `U`.

When comparing call or construct signatures, parameter names are ignored and rest parameters correspond to an unbounded expansion of optional parameters of the rest parameter element type. Note that specialized call and construct signatures (section 3.9.2.4) are not significant when determining subtype and supertype relationships. Also note that type parameters are not considered object types. Thus, the only subtypes of a type parameter `T` are `T` itself and other type parameters that are directly or indirectly constrained to `T`.

For our analysis, we need to make a few adjustments to this specification. First, the definition above does not deal with `strictNullChecks`, since this is the specification of TypeScript 1.8, because the specification for 2.0 was not yet available at the time of writing. Therefore, we must remove the third and fourth point, which prevent that `null` is a subtype of `string` for instance. In our analysis, we will also use number and boolean literal types. Similar to the sixth rule, we add two rules:

- `S` is a number literal type and `T` is the primitive type `number`.
- `S` is a boolean literal type and `T` is the primitive type `boolean`.

We will not consider excess properties, as these could be added in a type-checking phase. To reduce complexity, we do not support access modifiers (`private`, `protected`). We do not handle `Symbol` indexers (which are not part of specification of TypeScript 1.8 anyway). We also do not handle optional properties explicitly: an optional property of type `T` is modeled as `T | undefined`.

When comparing function types, all type parameters are replaced by `any`. That means that `<U, V>(u: U, v: V) => U` is assignable to `<U, V>(u: U, v: V) => V`, for instance. Even though this may give unexpected behavior, we use this definition in the analysis since it is easy to implement. Functions are bivariant, which means that `(a: "x") => void` is a subtype of `(a: string) => void`, which is a subtype of `(a: "y") => void`. However, `(a: "x") => void` is not a subtype of `(a: "y") => void`. We must change the subtype rules and require that the types of the arguments are supertypes (instead of subtypes or supertypes).

Types can have different representations. For instance, `{ x: number; } & { y: number; }` is equal to `{ x: number; y: number; }`. We say that two notations `T` and `U` represent the same type if `T` $\sqsubseteq$ `U` $\sqsubseteq$ `T`.

We add a new type, which represents a reference to a function with `never` as its only subtype, except for intersection types. It does not have supertypes, except for union types.

5.1.2 Lattice Definition

Definition 1 (Lattice)**.** *A lattice is a partially ordered set, where every two element have a unique least upper bound (join) and a unique lower bound (meet) (Nielson et al. 1999). For the ordering, $\sqsubseteq$, these axioms should hold for all x, y, z in the set:*

Reflexivity:

$$x \sqsubseteq x \tag{1}$$

Antisymmetry:

$$\textit{if } x \sqsubseteq y \textit{ and } y \sqsubseteq x \textit{ then } x = y \tag{2}$$

Transitivity:

$$\textit{if } x \sqsubseteq y \textit{ and } y \sqsubseteq z \textit{ then } x \sqsubseteq z \tag{3}$$

A lattice is bounded iff a greatest ($\top$) and least ($\bot$) element exist such that:

$$\forall x : \bot \sqsubseteq x \sqsubseteq \top \tag{4}$$

5.1.3 Lattice Axioms

The definition of the subset relation, yields a partial ordering on the set of types. We will prove the axioms here.

Lemma 1. *The subtype relation is reflexive.*

Proof. Reflexivity holds because of the first condition in the definition of the subset relation. $\square$

Lemma 2. *The subtype relation is antisymmetric.*

Proof. We defined that the same type can have different notations. When `T` $\sqsubseteq$ `U` $\sqsubseteq$ `T`, we say that `T` and `U` represent the same type. This is the exact condition for antisymmetry. $\square$

We will now prove that the relation is transitive for non-recursive types. We will first do that for primitive types, literal types and enum types. Afterwards we extend that with induction for union types, intersection types, object types and function types. In the following lemmas, `T` $\sqsubsetneq$ `U` $\sqsubsetneq$ `V` will be three types.

Lemma 3. *The subtype relation is transitive for all primitive types, literal types and enum types.*

Proof. Let `T`, `U`, `V` be three types from the set of primitive types, literal types and enum types. If `T` is `never` or `V` is `any`, transitivity holds trivially. So, we may assume that `T` $\neq$ `never` and `V` $\neq$ `any`

The only supertype of `any` is `any`. Given that `V` is not `any`, the other types will also not be `any`. By analogous reasoning, the same applies to `never`. Thus, we may assume that $\texttt{T}, \texttt{U}, \texttt{V} \notin \{\texttt{never}, \texttt{any}\}$.

If `T` is `number`, `boolean` and `string`, the only supertypes are `T` and `any`. Given that `U` is not `any` nor `T`, this case does not occur.

If `T` is a literal or enum type, then the only supertypes are `number`, `boolean` and `string`, other than `any`, union and intersection types. Thus, `U` is one of those. The only supertypes of `U` are `U` itself and `any`, thus this does not occur.

We have now considered all possibilities for `T` and we conclude that the subtype relation is transitive for all primitive types, literal types and enum types. $\square$

Lemma 4. *Let L be a set of type on which the transitivity axiom holds and K a type parameter constrained by a type in L. Then the subtype relation is transitive on $L \cup \{K\}$.*

Proof. If neither `T`, `U` nor `V` is the new type parameter `K`, this is trivial. Thus we assume that at least one of them is `K`. The supertypes are `K` and `any`. Thus, `T` and `U` cannot be a type parameter. Thus, `V` $=$ `K`. The subtypes of `K` are the type parameter itself and subtypes of its constraint. Let `C` be the constraint, which is in L. Then `T` $\sqsubseteq$ `U` $\sqsubseteq$ `C`. Given that $\texttt{T}, \texttt{U}, \texttt{C} \in L$, we know that `T` $\sqsubseteq$ `C`. $\square$

Lemma 5. *Let L be a set of types for which the transitivity axiom holds. Then the subtype relation is transitive for all types in L combined with object types and function types whose members have types in L.*

Proof. We can compare the subtype relation for all properties and indexers of objects, since these are in L. Recall from the 1.8 language specification (Microsoft 2016c, as reproduced from section 3.11.3) the following about functions and call signatures:

i. the signatures are of the same kind (call or construct),
ii. `M` has a rest parameter or the number of non-optional parameters in `N` is less than or equal to the total number of parameters in `M`,

iii. for parameter positions that are present in both signatures, each parameter type in `N` is a subtype or supertype of the corresponding parameter type in `M`, and
iv. the result type of `M` is `void`, or the result type of `N` is a subtype of that of `M`.

The first condition holds here. The second too because "less than or equal" is reflexive too. The last two points can again be proven because the parameter types and return type are in L. ☐

Lemma 6. *Let L be a set of types for which the transitivity axiom holds. Then the subtype relation is transitive for all types in L combined with intersection types whose parts are in L.*

Proof. We will now consider intersection types. Let `T`, `U`, `V` be intersection types of at least one part. Types that are not intersection types can be represented as an intersection of only one type. We denote the parts as $\mathtt{T}_i$. The subtype relation has two rules where the right side is an intersection type. The first says that each part of the intersection should be a supertype of the type on the left; the second says that the left side is an object type whose properties are present in the intersection type. The second rule is used to establish that `{ a, b }` $\sqsubseteq$ `{ a } & { b }`. Consider that only the first rule is used. Following the subtype rules:

$$\begin{aligned} &\mathtt{T} \sqsubseteq \mathtt{U} \\ \iff &\forall j : \mathtt{T} \sqsubseteq \mathtt{U}_j \\ \iff &\forall j : \exists i : \mathtt{T}_i \sqsubseteq \mathtt{U}_j \end{aligned} \tag{5}$$

Given that i depends on j, we will write it as $i(j)$. Using a similar derivation, it follows that $\forall l : \mathtt{U}_{k(l)} \sqsubseteq \mathtt{V}_l$ and that we must show that $\forall q : \mathtt{T}_{p(q)} \sqsubseteq \mathtt{V}_p$. Notice that $\forall l : \mathtt{T}_{i(k(l))} \sqsubseteq \mathtt{U}_{k(l)} \sqsubseteq \mathtt{U}_l$. Because the parts of the intersection types are in L, we know that $\forall l : \mathtt{T}_{i(k(l))} \sqsubseteq \mathtt{U}_l$. Thus we choose $p = k \cdot l$ and the theorem holds for intersection types.

If $\mathtt{T} \sqsubseteq \mathtt{U}$ uses the second rule for intersection types, then `T` is an intersection type of only one part. Given that a supertype will have at least all properties of the subtype, `V` will have all properties of `U` and the types of the properties of `V` will be supertypes of those of `U`. Thus, $\mathtt{T} \sqsubseteq \mathtt{V}$. The same can be shown for the case where $\mathtt{U} \sqsubseteq \mathtt{V}$ uses the second rule. ☐

Lemma 7. *Let L be a set of types for which the transitivity axiom holds. Then the subtype relation is transitive for all types in L combined with union types whose parts are in L.*

Proof. The theorem can be proven in almost the same way as for intersection types. We can expand the subtype relation for union types:

$$\begin{aligned} &\mathtt{T} \sqsubseteq \mathtt{U} \\ \iff &\forall i : \mathtt{T}_i \sqsubseteq \mathtt{U} \\ \iff &\forall i : \exists j : \mathtt{T}_i \sqsubseteq \mathtt{U}_j \end{aligned} \tag{6}$$

This gives that for each i, we can find j and k such that $\mathtt{T}_i \sqsubseteq \mathtt{U}_j \sqsubseteq \mathtt{U}_k$, thus $\mathtt{T}_i \sqsubseteq \mathtt{U}_k$ (using that the union parts are in L). ☐

We conclude that the subtype relation is transitive for all non-recursive types. Given that the analysis does not support recursive types, we do not have to consider those.

Lemma 8. *The subset relation gives a partial ordering on the set of non-recursive types.*

Proof. The previous lemmas are sufficient to show that the subtype relation is transitive on the set of all non-recursive types. ☐

We define the join (least upper bound, $\vee$) as follows:

$$a \vee b = a \mid b = a \cup b \tag{7}$$

We define the meet (greatest lower bound, $\wedge$):

$$a \wedge b = a \mathbin{\&} b = \{u \mathbin{\&} v : u \in a, v \in b\} \tag{8}$$

The join and meet are per definition of intersection and union types the least upper bound and greatest lower bound. To reduce the size of intersection and union types, we apply subtype reduction. We replace a union `T | U` by `T` if $\mathtt{U} \sqsubseteq \mathtt{T}$ and we replace `T & U` by `T` if $\mathtt{T} \sqsubseteq \mathtt{U}$. This means that `any` disappears in an intersection type and `never` in a union type. A union type that includes `any` becomes `any` and an intersection with `never` is replaced by `never`. Furthermore, an intersection type that has no assignable values, such as `string & number`, is replaced by `never`.

Lemma 9. *The type lattice is bounded*

Proof. Since `any` is a supertype of all types except for function references and the program can contain only a finite number of function declarations, $\top = \mathtt{any}|f_0|f_1|...|f_n$ is a supertype of all types, where $f_0...f_n$ represent the function references in the program. Given that `never` is an empty union type, it is the subtype of all types, so $\bot = \mathtt{never}$. Thus, the lattice is bounded. ☐

When combining the lemmas in this section, we find that the subtype relation on the set of types yields a bounded lattice.

5.1.4 Ascending Chain Condition

To have termination guarantees for our monotone framework our lattice should satisfy the *ascending chain condition*. The condition says that for each chain $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq ...$, there exists an $n \in \mathbb{N}$, such that $a_n = a_{n+1} = a_{n+2} = ...$ The definition of the type lattice does not satisfy this condition. For instance, the chain $a_i = 0 \mid 1 \mid 2 \ldots i$ will never terminate to a fixed point. Therefore, we apply *widening* to guarantee termination of the analysis.

We add three restrictions to our types: we restrict the size of union type, the number of properties, and the depth of nesting of object types (including arrays) (Fritz and Hage 2014). Let k be the chosen maximum size of a union type. Given that a finite number of function references exist, we can ignore these references in the widening of a union type. If a union type has a larger size, we widen in two phases: first, we replace all literal types by their primitive type. If the resulting union type is still larger than k, we replace it with `any`. That means that in the following example, the type of `i` in the body of the loop would be `0 | 1 | 2 ... n` if $n \leqslant k$, otherwise `number`:

```
for (let i = 0; i <= n; i++) { i; }
```

The two other restrictions are simpler to enforce: an object that is nested too deeply, or that has too many properties is replaced by `any`.

5.2 Transfer Functions

For each program point l, a monotone function exists that maps the incoming environment ($A_\circ$) to the output environment ($A_\bullet$):

$$A_\bullet = f_l(A_\circ) \tag{9}$$

Below we discuss the transfer functions for various kinds of program points.

5.2.1 Assignments and Variable Declarations

The transfer function for assignments replaces the previous type of the assigned variable with the type of the assigned value. The transfer function of a variable declaration sets the type of the variable to the initializer if present. If no initializer is present, it sets the type to `undefined` if the variable was block-scoped. A function-scoped variable may be used before its definition and the declaration does not change its value, so no type should be changed.

5.2.2 Type Guards

The transfer function for a type guard changes the type of a variable, similar to an assignment. The location of a type guard is defined as a tuple consisting an expression, and a flag that says whether it flows to the true- or false-branch. It will resolve the type of the expression and narrow it to a truthy (true when coerced to a boolean) or falsy (false when coerced) value. If that type is `never`, the following code is unreachable. Otherwise, if the expression is a reference to a variable, the type of the variable is narrowed to the calculated type. If the expression contains some other type guard, the referenced value is narrowed by the type guard.

5.2.3 Accessing Identifiers

In other transfer functions, the type of a variable is needed. In these cases, the type of the variable in a previous environment is needed instead of the current one. Assignments or type guards could exist between the previous and current environment. In `let x = 1; let y = x + (x = 2)`. When analyzing the type of `y`, the desired type is 3. This can be implemented by desugaring the program to static single assignment (SSA) form. An alternative is to take this into account in the control flow graph and to store the types of previous occurrences of a variable in the environment. We have implemented the latter. The transfer function for identifiers stores the type of the variable at that location in the environment. As an optimization, the type of a non-live variable could be cleared; otherwise it may propagate uselessly through a big part of the program.

5.2.4 Function Calls and Return Statements

When a call expression invokes a function reference, a new edge is inserted from the call expression to the entry point of the function and for each exit point of the function, an edge is created back to the call site. The function is analyzed for each calling site. This means that the analysis is context-sensitive. E.g., when an identity function is called from multiple locations, their return types are kept separately.

The transfer function for a return statement sets the type of the related call expression in the environment. It joins the environment before the call expression and the return statement, since a part of the variables in scope might originate from a different function call.

5.3 Standard Library

The definitions from the standard library are already written as TypeScript definitions, since the standard TypeScript compiler uses them. My implementation based on monotone frameworks can import this file too. However, since this file is approximately 19000 lines long and contains a lot of definitions, it was necessary to store this information outside of the environment. Declarations of the standard library could also be narrowed or reassigned and in these cases, the declaration is moved to the environment.

6. Benchmarks

In this section we provide some preliminary benchmarks of the performance and precision of our analysis.

For our benchmarks we have used a machine with the following specifications: MacBook Pro (Retina, 13-inch, Mid 2014), 2,8 GHz Intel Core i5, 16 GB 1600 MHz DDR3. We have run our implementations on two systems: `timeago.js` (339 lines of code) and `mustache.js` (625 lines).

We measured the running time of the implementations and the percentage of types that are `any`, as an indication of the precision. The types of identifiers, property access with dot or brackets and the `this` keyword were used in the measurements. Note that the used examples are JavaScript, thus lacking type annotations. For a fair comparison, we used the types from the TypeScript compiler only from code sections that were reachable according to the monotone implementation, such that the same code sections were analyzed. The full sources and resolved types can be found in (de Wolff 2016a) in the directory `cases`.

For the smaller of the two applications we see a large discrepancy in the timings: the interprocedural version is much, much slower. This seems to be due to a large number of function calls. Indeed, if we turn context-sensitivity off, the time is much reduced. The inter-procedural analysis does seem to give more accurate results. We analyzed the testcase with one library call (first table) and with tens of different calls (second table).

Implementation	% `any`	time
Inter-procedural monotone	11%	26.77s
Context insensitive monotone	15%	3.30s
TypeScript Compiler (2.0)	N/A	1.17s
TypeScript Compiler (2.1RC)	55%	1.11s

Inter-procedural monotone	28%	256.35s
Context insensitive monotone	36%	42.11s
TypeScript Compiler (2.0)	N/A	1.11s
TypeScript Compiler (2.1RC)	65%	1.14s

For the larger project, `mustache.js` the timings and precision measurements are very close to each other:

Inter-procedural monotone	21%	2.29s
Context insensitive monotone	21%	2.28s
TypeScript Compiler (2.0)	N/A	1.56s
TypeScript Compiler (2.1RC)	63%	1.13s

We note that we have also compiled the programs with TypeScript 1.8. That compiler is faster than the one for 2.0, but it is impossible to tell to which extent our control-flow based implementation is responsible, since many more things have changed between the two compilers. This is why we have left them out at this time. A nice additional result of the research was that we found two issues in the code of timeago.

7. Use Cases

Type guards have several use cases. They give more accurate types, which gives advantages in several scenarios. This section will demonstrate some of them.

7.1 Function Overloading

TypeScript does not support compile time function overloading. It is possible to create a function that has different signatures, but the types of the arguments have to be checked by the programmer. He can use type guards to do that. It is common in JavaScript to write a function that accepts either one value, or a list of values. Gulp, a build system for JavaScript, has a function that takes either one or a list of file names or patterns, and returns a stream of matched files. Such a function can be defined as follows:

```
function src(files: string | string[]) {
  if (typeof files === "string") {
    files = [files];
  }
  // files: string[];
}
```

Our monotone framework implementation can determine that at the end of `src`, `files` can only contain a value of type `string[]`.

```
type Tree<T, S> = Leaf<S> | Node<T, S>;
class Leaf<S> {
  constructor(public data: S) {}
}
class Node<T, S> {
  constructor(
    public item: T,
    public left: Tree<T, S>,
    public right: Tree<T, S>) {}
}
function map<T, S, U, V>(
    tree: Tree<T, S>,
    mapLeaf: (data: S) => V,
    mapNode: (item: T) => U
  ): Tree<U, V> {
  if (tree instanceof Leaf) {
    return new Leaf(mapLeaf(tree.data));
  } else {
    const { item, left, right } = tree;
    return new Node(mapNode(item),
      map(left, mapLeaf, mapNode),
      map(right, mapLeaf, mapNode));
  }
}
```

Figure 9. Type guards to mimick pattern matching on ADTs

7.2 Alternative to ADT and Pattern Matching

TypeScript does not have algebraic data types (ADT) nor pattern matching. Type guards offer an alternative to pattern matching. The code in Figure 9 demonstrates this. Note that prefixing a constructor argument with `public` makes it available as a property.

The `Leaf` class can also be replaced by `null` or `undefined` if a leaf contains no data. If another function only works on a `Node`, there is no need to create another type for it. For instance, a function that returns the leftmost element of a tree, can take `Node<T, S>` as argument. As an alternative it could also return `T | undefined`. This example can also be written with strings as tags, as shown in Figure 10.

Another use case is the return type of a generator. JavaScript supports generator functions which can stream results instead of returning a single value: multiple values can be pushed with `yield` and the final return value comes from a `return` statement. When consuming a generator, you get objects that have the yielded or returned value. These objects have a `done` property that is `false` if this value originates from a `yield` expression, and `true` if it originates from a `return` statement. TypeScript did not consider return values originally, since these are typically not used. This type can now be defined using a union type, which can be narrowed based on discriminated union types.

```
type IteratorResult<TYield, TReturn>
  = { done: false; value: TYield }
  | { done: true;  value: TReturn };
```

In a strongly typed functional language like Haskell, the functions `head` and `tail` are not defined for empty lists, causing the functions to crash with a run-time error when you call them with an empty list. In TypeScript, the parts of a union type are types too, so these can be used in the signature of these functions. Type guards can then be used to call the function safely. The compiler can check this and prevent runtime errors.

7.3 Null and Undefined Checks

When we were working on the control flow based implementation, Anders Hejlsberg of the TypeScript team was working on nullable types. Since version 2.0, `null` and `undefined` have become two

```
type Tree<T, S> = Leaf<S> | Node<T>;
interface Leaf<S> {
  kind: "leaf";
  data: S;
}
interface Node<T, S> {
  kind: "node";
  item: T;
  left: Tree<T, S>;
  right: Tree<T, S>;
}
function map<T, S, U, V>(
    tree: Tree<T, S>,
    mapLeaf: (data: S) => V,
    mapNode: (item: T) => U
  ): Tree<U, V> {
  if (tree.kind === "leaf") {
    return { kind: "leaf",
      data: mapLeaf(tree.data) };
  } else {
    const { item, left, right } = tree;
    return { kind: "node",
      item: mapNode(item),
      left: map(left, mapLeaf, mapNode),
      right: map(right, mapLeaf, mapNode) };
  }
}
```

Figure 10. Distinguishing node types by string tags

new types, and other types do not contain these values any more. Using the control flow based type analysis, the compiler can give accurate warnings when a variable can be `null` or `undefined`. A variable that contains a string or `null` can be annotated with `string | null`. For cases where the compiler is too restrictive, a new type cast was introduced: `x!` removes `null` and `undefined` from the type of `x`.

8. Related Work

Considerable work has been done on refinement types. Guha et al. (Guha et al. 2011) have investigated refinement types in scripting languages, including JavaScript, Python and Ruby. They analyzed a code base of 617,666 lines of code, which contained 3,298 `undefined` and `null` checks, 17 `instanceof` checks and 474 `typeof` checks. (Actual numbers can be higher because of conservative checks in their analysis. For more recent code, these numbers might be different since the 2016 release of ECMAScript adds classes.) They introduce a core calculus λ_S to model scripting languages, including JavaScript, with type checking using type guards.

Chugh et al. (Chugh et al. 2012) have introduced another core calculus, System D, supporting nested refinement types and have proven its soundness. Their analysis supports higher-order functions, polymorphism and dictionaries.

Only few research projects related to TypeScript are available at the moment. Bierman et al. (Bierman et al. 2014) give an introduction to TypeScript and its type system. However, because of the rapid development of TypeScript, this overview is somewhat out-dated. Rastogi et al. (Rastogi et al. 2015) give an overview of Safe TypeScript, a fork that adds runtime checks in the generated JavaScript code, to overcome the unsoundness of TypeScript.

More information on the TypeScript language can be found in the TypeScript Documentation (Microsoft 2016a). TypeScript Blueprints (de Wolff 2016c) gives a practical introduction to TypeScript for JavaScript programmers.

Flow (Facebook 2016) is a static type checker for JavaScript. It only functions as a type checker and not as a complete compiler. Flow focuses more on soundness, whereas TypeScript focuses on developer tools such as editor support.

9. Conclusion

This paper presents two distinct implementations of type guards in the context of TypeScript. We started by elucidating the shortcomings of the existing syntax-directed implementation, and discussed a control flow based implementations that has approximately the same performance, but better accuracy. Both versions are suitable for real-time usage in editors, since they do not require a whole program analysis. The control flow based implementation has replaced the syntax directed implementation in the TypeScript compiler, thanks to additional work by Anders Hejlsberg. Type guards facilitate the checking of nullable types; the analysis can give accurate information whether a variable might be `undefined` or `null` (Rosenwasser 2016).

An implementation based on monotone frameworks is discussed for cases where a whole program analysis is appropriate. This gives more accurate types. Type guards can increase the accuracy of the types when a program has polymorphic functions.

9.1 Future Work

All discussed implementations cannot see dependencies between types. The standard library of JavaScript has a dictionary class, `Map`, which has a function `has` to check whether a key binding exists, and a function `get` to retrieve the associated value. If `get` is called with a key that does not exist, it returns `undefined`. The return type of `get` on a `Map<T, V>`, where `T` is the type of the key and `V` the type of the value, is defined as `V | undefined`. Ideally, the call to `has` would be a type guard for `get`. The following code block demonstrates this:

```
const map: Map<string, number> = ...;
const key: string = ...;
if (map.has(key)) { const value = map.get(key); }
```

According to our implementations, the type of `value` is `number | undefined`. The type guard `map.has(key)` creates a dependency between the types of `map` and `key`, which cannot be tracked by these implementations; this is by no means the only such case.

Another idea is to compute a greatest fixed point for the monotone framework. One advantage is that a greatest fixed point computation starts with a sound solution working towards a more precise one. In a real-time setting this may be advantageous, improving the approximations with a given upper bound to the time we may spent doing so, to guarantee responsiveness. Moreover, for variables that have an annotation, we can start the analysis by setting the type of that variable to the annotation, instead of `any`.

References

G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.

R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103686. URL `http://doi.acm.org/10.1145/2103656.2103686`.

I. G. de Wolff. Monotone type checker. `https://github.com/ivogabe/monotone-type-checker`, 2016a. Accessed: 20-7-2016.

I. G. de Wolff. Control flow based type guards - pull request 6959. `https://github.com/Microsoft/TypeScript/pull/6959`, 2016b. Accessed: 20-7-2016.

I. G. de Wolff. *TypeScript Blueprints*. PACKT Publishing, 2016c.

Facebook. Flow — a static type checker for javascript. `https://flowtype.org`, 2016.

L. Fritz and J. Hage. Cost versus precision for approximate typing for python. Technical Report UU-CS-2014-017, Department of Information and Computing Sciences, Utrecht University, 2014.

A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In G. Barthe, editor, *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, pages 256–275, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19718-5.

A. Hejlsberg. Control flow based type analysis - pull request 8010. `https://github.com/Microsoft/TypeScript/pull/8010`, 2016a. Accessed: 20-7-2016.

A. Hejlsberg. Number, enum, and boolean literal types - pull request 9407. `https://github.com/Microsoft/TypeScript/pull/9407`, 2016b. Accessed: 6-7-2016.

Microsoft. Quick start - typescript. `https://www.typescriptlang.org/docs/`, 2016a. Accessed: 28-9-2016.

Microsoft. Typescript design goals. `https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals`, 2016b. Accessed: 12-7-2016.

Microsoft. Typescript language specification version 1.8. `https://github.com/Microsoft/TypeScript/blob/29985f33b7cabf9f549721c368ba2118d147779f/doc/TypeScriptLanguage\%20Specification.pdf`, 2016c. Accessed: 4-7-2016.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676971. URL `http://doi.acm.org/10.1145/2676726.2676971`.

D. Rosenwasser. Announcing typescript 2.0 beta, 2016.

J. Turner. Typescript and the road to 2.0. `https://blogs.msdn.microsoft.com/typescript/2014/10/22/typescript-and-the-road-to-2-0/`, 10 2014. Accessed: 6-7-2016.

J. Turner. Angular 2: Built on typescript. `https://blogs.msdn.microsoft.com/typescript/2015/03/05/angular-2-built-on-typescript/`, 3 2015. Accessed: 6-7-2016.