The Template Type Checking Engine

The `typecheck` package is concerned with template type-checking, the process by which the compiler determines and understands the TypeScript types of constructs within component templates. It's used to perform actual type checking of templates (similarly to how TypeScript checks code for type errors). It also provides the `TemplateTypeChecker` API which is a conceptual analogue to TypeScript's own `ts.TypeChecker`, exposing various semantic details about template types to consumers such as the Angular Language Service.

The template type-checking engine is complex, as TypeScript itself is not very pluggable when it comes to the type system. The main algorithm for template type-checking is as follows:

1. The input `ts.Program` is analyzed, and information about directives/pipes as well as candidate templates is collected.
2. Each candidate template is converted into a "type-check block", or TCB, a TypeScript function that semantically describes the operations in the template as well as their types.
3. A derivative `ts.Program` is created from the user's input program from step 1, plus the newly generated TCBs.
4. TypeScript is asked to produce diagnostics for the TCBs, which arise from type errors within the template.
5. TCB diagnostics are converted to template diagnostics and reported to the user.

This algorithm relies extensively on TypeScript's ability to rapidly type check incremental changes in a `ts.Program` for its performance characteristics. Much of its design is optimized to ensure TypeScript has to do the minimum incremental work to check the new `ts.Program`.

Type Check Blocks

To understand and check the types of various operations and structures within templates, the `typecheck` system maps them to TypeScript code, encoding them in such a way as to express the intent of the operation within the type system.

TCBs are not ever emitted, nor are they referenced from any other code (they're unused code as far as TypeScript is concerned). Their *runtime* effect is therefore unimportant. What matters is that they express to TypeScript the type relationships of directives, bindings, and other entities in the template. Type errors within TCBs translate directly to type errors in the original template.

Theory

Given a component `SomeCmp`, its TCB takes the form of a function:

```
function tcb(ctx: SomeCmp): void {
  // TCB code
```

```
}
```

Encoding the TCB as a function serves two purposes:

1. It provides a lexical scope in which to declare variables without fear of collisions.
2. It provides a convenient location (the parameter list) to declare the component context.

The component context is the theoretical component instance associated with the template. Expressions within the template often refer to properties of this component.

For example, if `SomeCmp`'s template has an interpolation expression `{{foo.bar}}`, this suggests that `SomeCmp` has a property `foo`, and that `foo` itself is an object with a property `bar` (or in a type sense, that the type of `SomeCmp.foo` has a property `bar`).

Such a binding is expressed in the TCB function, using the `ctx` parameter as the component instance:

```
function tcb(ctx: SomeCmp): void {
  '' + ctx.foo.bar;
}
```

If `SomeCmp` does not have a `foo` property, then TypeScript will produce a type error/diagnostic for the expression `ctx.foo`. If `ctx.foo` does exist, but is not of a type that has a `bar` property, then TypeScript will catch that too. By mapping the template expression `{{foo.bar}}` to TypeScript code, the compiler has captured its *intent* in the TCB in a way that TypeScript can validate.

Types of template declarations Not only can a template consume properties declared from its component, but various structures within a template can also be considered "declarations" which have types of their own. For example, the template:

```
<input #name>
{{name.value}}
```

declares a single `<input>` element with a local ref `#name`, meaning that within this template `name` refers to the `<input>` element. The `{{name.value}}` interpolation is reading the `value` property of this element.

Within the TCB, the `<input>` element is treated as a declaration, and the compiler leverages the powerful type inference of `document.createElement`:

```
function tcb(ctx: SomeCmp): void {
  var _t1 = document.createElement('input');
  '' + _t1.value;
}
```

The `_t1` variable represents the instance of the `<input>` element within the template. This statement will never be executed, but its initialization expression is used to infer a correct type of `_t1` (`HTMLInputElement`), using the `document.createElement` typings that TypeScript provides.

By knowing the type of this element, the expression involving the `name` local ref can be translated into the TCB as `_t1.value`. TypeScript will then validate that `_t1` has a `value` property (really, that the `HTMLInputElement` type has a `value` property).

Directive types Just like with HTML elements, directives present on elements within the template (including those directives which are components) are treated as declarations as well. Consider the template:

```
<other-cmp [foo]="bar"></other-cmp>
```

The TCB for this template looks like:

```
function tcb(ctx: SomeCmp): void {
  var _t1: OtherCmp = null!;
  _t1.foo = ctx.bar;
}
```

Since `<other-cmp>` is a component, the TCB declares `_t1` to be of that component's type. This allows for the binding `[foo]="bar"` to be expressed in TypeScript as `_t1.foo = ctx.bar` - an assignment to `OtherCmp`'s `@Input` for `foo` of the `bar` property from the template's context. TypeScript can then type check this operation and produce diagnostics if the type of `ctx.bar` is not assignable to the `_t1.foo` property which backs the `@Input`.

Generic directives & type constructors The above declaration of `_t1` using the component's type only works when the directive/component class is not generic. If `OtherCmp` were declared as:

```
export class OtherCmp<T> {
  ...
}
```

then the compiler could not write

```
var _t1: OtherCmp<?> = ...;
```

without picking a value for the generic type parameter `T` of `OtherCmp`. How should the compiler know what type the user intended for this component instance?

Ordinarily, for a plain TypeScript class such as `Set<T>`, the generic type parameters of an instance are determined in one of two ways:

1. Directly, via construction: `new Set<string>()`

2. Indirectly, via inference from constructor parameters: `new Set(['foo', 'bar']); // Set<string>`

For directives, neither of these options makes sense. Users do not write direct constructor calls to directives in a template, so there is no mechanism by which they can specify generic types directly. Directive constructors are also dependency-injected, and generic types cannot be used as DI tokens and so they cannot be inferred from DI.

Instead, conceptually, the generic type of a directive depends on the types bound to its *inputs*. This is immediately evident for a directive such as `NgFor`:

```
@Directive({selector: '[ngFor]'})
export class NgFor<T> {
  @Input() ngForOf!: Iterable<T>;
}
```

(Note: the real `NgFor` directive is more complex than the simplistic version examined here, but the same principles still apply)

In this case, the `T` type parameter of `NgFor` represents the value type of the `Iterable` over which we're iterating. This depends entirely on the `Iterable` passed to `NgFor` - if the user passes a `User[]` array, then this should conceptually create an `NgFor<User>` instance. If they pass a `string[]` array, it should be an `NgFor<string>` instead.

To infer a correct type for a generic directive, the TCB system generates a **type constructor** for the directive. The type constructor is a "fake" constructor which can be used to infer the directive type based on any provided input bindings.

A type constructor for the simplistic `NgFor` directive above would look like:

```
declare function ctor1<T>(inputs: {ngForOf?: Iterable<T>}): NgFor<T>;
```

This type constructor can then be used to infer the instance type of a usage of `NgFor` based on provided bindings. For example, the template:

```
<div *ngFor="let user of users">...</div>
```

Would use the above type constructor in its TCB:

```
function tcb(ctx: SomeCmp): void {
  var _t1 = ctor1({ngForOf: ctx.users});
  // Assuming ctx.users is User[], then _t1 is inferred as NgFor<User>.
}
```

A single type constructor for a directive can be used in multiple places, whenever an instance of the directive is present in the template.

Nested templates & structural directives `NgFor` is a structural directive, meaning that it applies to a nested `<ng-template>`. That is, the template:

```
<div *ngFor="let user of users">
  {{user.name}}
</div>
```

is syntactic sugar for:

```
<ng-template ngFor let-user="$implicit" [ngForOf]="users">
  <div>{{user.name}}</div>
</ng-template>
```

The `NgFor` directive injects a `TemplateRef` for this *embedded view*, as well as a `ViewContainerRef`, and at runtime creates dynamic instances of this nested template (one per row). Each instance of the embedded view has its own "template context" object, from which any `let` bindings are read.

In the TCB, the template context of this nested template is itself a declaration with its own type. Since a structural directive can in theory create embedded views with any context object it wants, the template context type always starts as `any`:

```
declare function ctor1(inputs: {ngForOf?: Iterable<T>}): NgFor<T>;

function tcb(ctx: SomeCmp): void {
  // _t1 is the NgFor directive instance, inferred as NgFor<User>.
  var _t1 = ctor1({ngForOf: ctx.users});

  // _t2 is the context type for the embedded views created by the NgFor structural directi
  var _t2: any;

  // _t3 is the let-user variable within the embedded view.
  var _t3 = _t2.$implicit;

  // Represents the `{{user.name}}` interpolation within the embedded view.
  '' + _t3.name;
}
```

Note that the `any` type of the embedded view context `_t2` effectively disables type-checking of the `{{user.name}}` expression, because the compiler has no idea what type `user` will be when `NgFor` instantiates this embedded view. Since this instantiation is imperative code, the compiler cannot know what `NgFor` will do.

Template context hints To solve this problem, the template type-checking engine allows structural directives to *declaratively* narrow the context type of any embedded views they create. The example `NgFor` directive would do this by declaring a static `ngTemplateContextGuard` function:

```ts
@Directive({selector: '[ngFor]'})
export class NgFor<T> {
  @Input() ngForOf!: Iterable<T>;

  static ngTemplateContextGuard<T>(dir: NgFor<T>, ctx: any): ctx is NgForContext<T> {
    return true; // implementation is not important
  }
}

export interface NgForContext<T> {
  $implicit: T;
}
```

The `typecheck` system is aware of the presence of this method on any structural directives used in templates, and uses it to narrow the type of its context declarations. So with this method on `NgFor`, the template from before would now generate a TCB of:

```ts
declare function ctor1(inputs: {ngForOf?: Iterable<T>}): NgFor<T>;

function tcb(ctx: SomeCmp): void {
  // _t1 is the NgFor directive instance, inferred as NgFor<User>.
  var _t1 = ctor1({ngForOf: ctx.users});

  // _t2 is the context type for the embedded views created by the NgFor structural directi
  var _t2: any;

  if (NgFor.ngTemplateContextGuard(_t1, _t2)) {
    // NgFor's ngTemplateContextGuard has narrowed the type of _t2
    // based on the type of _t1 (the NgFor directive itself).
    // Within this `if` block, _t2 is now of type NgForContext<User>.

    // _t3 is the let-user variable within the embedded view.
    // Because _t2 is narrowed, _t3 is now of type User.
    var _t3 = _t2.$implicit;

    // Represents the `{{user.name}}` interpolation within the embedded view.
    '' + _t3.name;
  }
}
```

Because the `NgFor` directive *declared* to the template type checking engine what type it intends to use for embedded views it creates, the TCB has full type information for expressions within the nested template for the `*ngFor` invocation, and the compiler is able to correctly type check the `{{user.name}}` expression.

'binding' guard hints `NgIf` requires a similar, albeit not identical, operation to perform type narrowing with its nested template. Instead of narrowing the template context, `NgIf` wants to narrow the actual type of the expression within its binding. Consider the template:

```
<div *ngIf="user !== null">
  {{user.name}}
</div>
```

Obviously, if `user` is potentially `null`, then this `NgIf` is intended to only show the `<div>` when `user` actually has a value. However, from a type-checking perspective, the expression `user.name` is not legal if `user` is potentially `null`. So if this template was rendered into a TCB as:

```
function tcb(ctx: SomeCmp): void {
  // Type of the NgIf directive instance.
  var _t1: NgIf;

  // Binding *ngIf="user != null".
  _t1.ngIf = ctx.user !== null;

  // Nested template interpolation `{{user.name}}`
  '' + ctx.user.name;
}
```

Then the `'' + user.name` line would produce a type error that `user` might be `null`. At runtime, though, the `NgIf` prevents this condition by only instantiating its embedded view if its bound `ngIf` expression is truthy.

Similarly to `ngTemplateContextGuard`, the template type checking engine allows `NgIf` to express this behavior by adding a static field:

```
@Directive({selector: '[ngIf]'})
export class NgIf {
  @Input() ngIf!: boolean;

  static ngTemplateGuard_ngIf: 'binding';
}
```

The presence and type of this static property tells the template type-checking engine to reflect the bound expression for its `ngIf` input as a guard for any embedded views created by the structural directive. This produces a TCB:

```
function tcb(ctx: SomeCmp): void {
  // Type of the NgIf directive instance.
  var _t1: NgIf;

  // Binding *ngIf="user != null".
  _t1.ngIf = ctx.user !== null;
```

```
  // Guard generated due to the `ngTemplateGuard_ngIf` declaration by the NgIf directive.
  if (user !== null) {
    // Nested template interpolation `{{user.name}}`.
    // `ctx.user` here is appropriately narrowed to be non-nullable.
    '' + ctx.user.name;
  }
}
```

The guard expression causes TypeScript to narrow the type of `ctx.user` within the `if` block and identify that `ctx.user` cannot be `null` within the embedded view context, just as `NgIf` itself does during rendering.

Generation process

Angular templates allow forward references. For example, the template:

```
The value is: {{in.value}}
<input #in>
```

contains an expression which makes use of the `#in` local reference before the targeted `<input #in>` element is declared. Since such forward references are not legal in TypeScript code, the TCB may need to declare and check template structures in a different order than the template itself.

Two phase generation To support this out-of-order generation, the template type checking engine processes templates using an abstraction of known as a `TcbOp`, or TCB operation. `TcbOp`s have two main behaviors:

1. Executing a `TcbOp` appends code to the TCB being generated.
2. Executing a `TcbOp` optionally returns an identifier or expression, which can be used by other operations to refer to some aspect of the generated structure.

The main algorithm for TCB generation then makes use of this abstraction:

1. The template is processed in a depth-first manner, and `TcbOp`s representing the code to be generated for the structures and expressions within are enqueued into a `TcbOp` queue.
2. Execution of operations begins from the start of the queue.
3. As each `TcbOp` is executed, its result is recorded.
4. An executing `TcbOp` may request the results of other `TcbOp`s for any dependencies, even `TcbOp`s which appear later in the queue and have not yet executed.
5. Such dependency `TcbOp`s are executed "on demand", when requested.

This potential out-of-order execution of `TcbOp`s allows for the TCB ordering to support forward references within templates. The above forward reference example thus results in a `TcbOp` queue of two operations:

```
[
  TcbTextInterpolationOp(`in.value`),
  TcbElementOp('<input #in>'),
]
```

Execution of the first `TcbTextInterpolationOp` will attempt to generate code representing the expression. Doing this requires knowing the type of the `in` reference, which maps to the element node for the `<input>`. Therefore, as part of executing the `TcbTextInterpolationOp`, the execution of the `TcbElementOp` will be requested. This operation produces TCB code for the element:

```
var t1 = document.createElement('input');
```

and returns the expression `t1` which represents the element type. The `TcbTextInterpolationOp` can then finish executing and produce its code:

```
'' + t1.value;
```

resulting in a final TCB:

```
var t1 = document.createElement('input');
'' + t1.value;
```

This ordering resolves the forward reference from the original template.

Tracking of `TcbOps` In practice, a `TcbOp` queue is maintained as an array, where each element begins as a `TcbOp` and is later replaced with the resulting `ts.Expression` once the operation is executed. As `TcbOp`s are generated for various template structures, the index of these operations is recorded. Future dependencies on those operations can then be satisfied by looking in the queue at the appropriate index. The contents will either be a `TcbOp` which has yet to be executed, or the result of the required operation.

`Scope` Angular templates are nested structures, as the main template can contain embedded views, which can contain their own views. Much like in other programming languages, this leads to a scoped hierarchy of symbol visibility and name resolution.

This is reflected in the TCB generation system via the `Scope` class, which actually performs the TCB generation itself. Each embedded view is its own `Scope`, with its own `TcbOp` queue.

When a parent `Scope` processing a template encounters an `<ng-template>` node:

1. a new child `Scope` is created from the nodes of the embedded view.
2. a `TcbTemplateBodyOp` is added to the parent scope's queue, which upon execution triggers generation of the child `Scope`'s TCB code and inserts it into the parent's code at the right position.

Resolution of names (such as local refs) within the template is also driven by the `Scope` hierarchy. Resolution of a name within a particular embedded view

begins in that view's `Scope`. If the name is not defined there, resolution proceeds upwards to the parent `Scope`, all the way up to the root template `Scope`.

If the name resolves in any given `Scope`, the associated `TcbOp` can be executed and returned. If the name does not resolve even at the root scope, then it's treated as a reference to the component context for the template.

Breaking cycles It's possible for a template to contain a referential cycle. As a contrived example, if a component is generic over one of its inputs:

```
<generic-cmp #ref [in]="ref.value"></generic-cmp>
```

Here, type-checking the `[in]` binding requires knowing the type of `ref`, which is the `<generic-cmp>`. But the `<generic-cmp>` type is inferred using a type constructor for the component which requires the `[in]` binding expression:

```
declare function ctor1<T>(inputs: {in?: T}): GenericCmp<T>;

function tcb(ctx: SomeCmp): void {
  // Not legal: cannot refer to t1 (ref) before its declaration.
  var t1 = ctor1({in: t1.value});

  t1.in = t1.value;
}
```

This is only a cycle in the *type* sense. At runtime, the component is created before its inputs are set, so no cycle exists.

To get around this, `TcbOp`s may optionally provide a fallback value via a `circularFallback()` method, which will be used in the event that evaluation of a `TcbOp` attempts to re-enter its own evaluation. In the above example, the `TcbOp` for the directive type declares a fallback which infers a type for the directive *without* using its bound inputs:

```
declare function ctor1<T>(inputs: {in?: T}): GenericCmp<T>;

function tcb(ctx: SomeCmp): void {
  // Generated to break the cycle for `ref` - infers a placeholder
  // type for the component without using any of its input bindings.
  var t1 = ctor1(null!);

  // Infer the real type of the component using the `t1` placeholder
  // type for `ref`.
  var t2 = ctor1({in: t1.value});

  // Check the binding to [in] using the real type of `ref`.
  t2.in = t2.value;
}
```

Optional operations Some `TcbOp`s are marked as optional. Optional operations are never executed as part of processing the op queue, and are skipped when encountered directly. However, other ops may depend on the results of optional operations, and the optional ops are then executed when requested in this manner.

The TCB node generated to represent a DOM element type (`TcbElementOp`) is an example of such an optional operation. Such nodes are only useful if the type of the element is referenced in some other context (such as via a `#ref` local reference). If not, then including it in the TCB only serves to bloat the TCB and increase the time it takes TypeScript to process it.

Therefore, the `TcbElementOp` is optional. If nothing requires the element type, it won't be executed and no code will be generated for the element node.

Source mapping and diagnostic translation

Once TCB code is generated, TypeScript is able to process it. One use case is to have TypeScript produce diagnostics related to type issues within the TCB code, which indicate type issues within the template itself. These diagnostics are of course expressed in terms of the TCB itself, but the compiler wants to report diagnostics to the user in terms of the original template text.

To do this, the template type checking system is capable of mapping from the TCB back to the original template. The algorithm to perform this mapping relies on the emission of source mapping information into the TCB itself, in the form of comments.

Consider a template expression of the form:

```
{{foo.bar}}
```

The generated TCB code for this expression would look like:

```
'' + ctx.foo.bar;
```

What actually gets generated for this expression looks more like:

```
'' + (ctx.foo /* 3,5 */).bar /* 3,9 */;
```

The trailing comment for each node in the TCB indicates the template offsets for the corresponding template nodes. If for example TypeScript returns a diagnostic for the `ctx.foo` part of the expression (such as if `foo` is not a valid property on the component context), the attached comment can be used to map this diagnostic back to the original template's `foo` node.

`TemplateId` As multiple TCBs can be present in a single typecheck file, an additional layer of mapping is necessary to determine the component and template for a given TCB diagnostic.

During TCB generation, a numeric `TemplateId` is assigned to each template declared within a given input file. These `TemplateId` are attached in a comment to the TCBs for each template within the corresponding typecheck files. So the full algorithm for mapping a diagnostic back to the original template involves two steps:

1. Locate the top level TCB function declaration that contains the TCB node in error, extract its `TemplateId`, and use that to locate the component and template in question.
2. Locate the closest source map comment to the TCB node in error, and combine that with knowledge of the template to locate the template node in error to produce the template diagnostic.

Ignore markers Occasionally, code needs to be generated in the TCB that should not produce diagnostics. For example, a safe property navigation operation `a?.b` is mapped to a TypeScript ternary operation `a ? a.b : undefined` (the actual code here may be more complex). If the original `a` expression is in error, then redundant diagnostics would be produced from both instances of `a` in the generated TCB code.

To avoid this, generated code can be marked with a special comment indicating that any diagnostics produced within should be ignored and not converted into template diagnostics.

Why not real sourcemaps? TypeScript unfortunately cannot consume sourcemaps, only produce them. Therefore, it's impossible to generate a source map to go along with the TCB code and feed it to TypeScript.

Generation diagnostics

Not all template errors will be caught by TypeScript from generated TCB code. The template type checking engine may also detect errors during the creation of the TCB itself. Several classes of errors are caught this way:

- DOM schema errors, like elements that don't exist or attributes that aren't correct.
- Missing pipes.
- Missing `#ref` targets.
- Duplicate `let-variable`s.
- Attempts to write to a `let-variable`.

These errors manifest as "generation diagnostics", diagnostics which are produced during TCB generation, before TCB code is fed to TypeScript. They're ultimately reported together with any converted TCB diagnostics, but are tracked separately by the type checking system.

Inline operations

In certain cases, generation of TCBs as separate, independent structures may not be possible.

Inline type constructors The mechanics of generic directive type constructors were described above. However, the example given was for a directive with an unbounded generic type. If the directive has a *bounded* generic type, then the type bounds must be repeated as part of the type constructor. For example, consider the directive:

```
@Directive({selector: '[dir]'})
export class MyDir<T extends string> {
  @Input() value: T;
}
```

In order to properly infer and check the type of this directive, the type constructor must include the generic bounds:

```
declare function ctor1<T extends string>(inputs: {value?: T}): MyDir<T>;
```

A generic bound for a type parameter is an arbitrary type, which may contain references to other types. These other types may be imported, or declared locally in the same file as a directive. This means that copying the generic bounds into the type constructor for the directive is not always straightforward, as the TCB which requires this type constructor is usually not emitted into the same file as the directive itself.

For example, copying the generic bounds is not possible for the directive:

```
interface PrivateInterface {
  field: string;
}

@Directive({selector: '[dir]'})
export class MyDir<T extends PrivateInterface> {
  @Input() value: T;
}
```

In such cases, the type checking system falls back to an alternative mechanism for declaring type constructors: adding them as static methods on the directive class itself. As part of the type checking phase, the above directive would be transformed to:

```
interface PrivateInterface {
  field: string;
}

@Directive({selector: '[dir]'})
```

```
export class MyDir<T extends PrivateInterface> {
  @Input() value: T;

  static ngTypeCtor<T extends PrivateInterface>(inputs: {value?: T}): MyDir<T> { return null
}
```

Putting the type constructor declaration within the directive class itself allows the generic signature to be copied without issue, as any references to other types in the same file will still be valid. The type constructor can then be consumed from a TCB as `MyDir.ngTypeCtor`. This is known as an "inline" type constructor.

Additions of such inline type checking code have significant ramifications on the performance of template type checking, as discussed below.

Inline Type Check Blocks A similar problem exists for generic components and the declaration of TCBs. A TCB function must also copy the generic bounds of its context component:

```
function tcb<T extends string>(ctx: SomeCmp<T>): void {
  /* tcb code */
}
```

If `SomeCmp`'s generic bounds are more complex and reference types that cannot be safely reproduced in the separate context of the TCB, then a similar workaround is employed: the compiler generates the TCB as an "inline" static method on the component.

This can also happen for components which aren't themselves importable:

```
it('should type-check components declared within functions', () => {
  @Component({
    selector: 'some-cmp',
    template: '{{foo.bar}}'})
  class SomeCmp {
    foo: Foo = {...};
  }
});
```

Such component declarations are still processed by the compiler and can still be type checked using inline TCBs.

Environment

TCBs are not standalone, and require supporting code such as imports to be properly checked. Additonally, multiple TCBs can share declarations regarding common dependencies, such as type constructors for directives as well as pipe instances.

Each TCB is therefore generated in the context of an `Environment`, which loosely represents the file which will ultimately contain the TCB code.

During TCB generation, the `Environment` is used to obtain references to imported types, type constructors, and other shared structures.

`TypeCheckingConfig` `Environment` also carries the `TypeCheckingConfig`, an options interface which controls the specifics of TCB generation. Through the `TypeCheckingConfig`, a consumer can enable or disable various kinds of strictness checks and other TCB operations.

The `TemplateTypeChecker`

The main interface used by consumers to interact with the template type checking system is the `TemplateTypeChecker`. Methods on this interface allow for various operations related to TCBs, such as:

- Generation of diagnostics.
- Retrieving `Symbol`s (the template equivalent to TypeScript's `ts.Symbol`) for template nodes.
- Retrieving TCB locations suitable for autocompletion operations.

Symbols

The TCB structures representing specific template nodes and operations are highly useful for "code intelligence" purposes, not just for type checking. For example, consider the "Go to Definition" function within an IDE. Within TypeScript code, TypeScript's language service can access semantic information about an identifier under the user's cursor, and locate the referenced declaration for that identifier.

Because TCBs are TypeScript code, the TypeScript language service can be used within TCB code to locate definitions in a similar manner. Such a "template language service" works in the following manner:

1. Find the template node under the user's cursor.
2. Locate its equivalent structure in the TCB.
3. Ask TypeScript's language service to perform the requested operation on the TCB node.

Step 2 in this algorithm is made possible by the `TemplateTypeChecker`'s APIs for retrieving `Symbol`s for template nodes. A `Symbol` is a structure describing the TCB information associated with a given template node, including positions within the TCB where type information about the node in question can be found.

For example, an `ElementSymbol` contains a TCB location for the element type, as well as any directives which may be present on the element. Such information can be used by a consumer to query TypeScript for further information about

the types present in the template for that element. This is used in the Angular Language Service to implement many of its features.

The type checking workflow

Like TypeScript's `ts.TypeChecker`, the `TemplateTypeChecker`'s operations are lazy. TCB code for a given template is only generated on demand, to satisfy a query for information about that specific template. Once generated, future queries regarding that template can be answered using the existing TCB structure.

The lazy algorithm used by the `TemplateTypeChecker` is as follows.

1. Given a query, determine which (if any) components need TCBs generated.
2. Request sufficient information about those components and any used directives/pipes from the main compiler.
3. Generate TCB source text.
4. Obtain a `ts.Program` containing the new, generated TCBs.
5. Use TypeScript APIs against this program and the TCBs to answer the query.

To enable the most flexibility for integrating the template type checking engine into various compilation workflows, steps 2 and 4 are abstracted behind interfaces.

Step 2: template information

To answer most queries, the `TemplateTypeChecker` will require the TCB for a specific component. If not already generated, this involves generating a TCB for that component, and incorporating that TCB somehow into a new `ts.Program`.

Each source file in the user's program has a corresponding synthetic "typecheck" file which holds the TCBs for any components it contains. Updating this typecheck file with new contents (a new TCB) has a large fixed cost, regardless of the scope of the textual change to the file. Thus, the `TemplateTypeChecker` will always generate *all* TCBs for a given input file, instead of just the one it needs. The marginal cost of generating the extra TCBs is extremely low.

To perform this generation, the type checking system first constructs a "context" into which template information can be recorded. It then requests (via an interface, the `ProgramTypeCheckAdapter`) that this context be populated with information about any templates present in the input file(s) it needs to process. Once populated, the information in the context can be used to drive TCB generation.

This process is somewhat complicated by the bookkeeping required to track which TCBs have been generated as well as the source mapping information for components within those TCBs.

On the compiler side, this interface drives the `TraitCompiler` to call the

`typeCheck()` method of the `ComponentDecoratorHandler` for each decorated component class.

Step 4: `ts.Programs`

The main output of the TCB generation mechanism is a list of new text contents for various source files. Typically this contains new TCB text for "typecheck" source files, but inline operations may require changes to actual user input files.

Before diagnostics can be produced or the `ts.TypeChecker` API can be used to interrogate types within the TCBs, these changes need to be parsed and incorporated into a `ts.Program`.

Creating such a `ts.Program` is not the responsibility of the `TemplateTypeChecker`. Instead, this creation is abstracted behind the `TypeCheckingProgramStrategy` interface. A strategy which implements this interface allows the type checking system to request textual changes be applied to the current `ts.Program`, resulting in an updated `ts.Program`.

As a convenience, the type-checking system provides an implementation of this abstraction, the `ReusedProgramStrategy`, which can be used by consumers that manage `ts.Program`s via TypeScript's `ts.createProgram` API. The main compiler uses this strategy to support template type checking.

`ts.Program`s can also be created via Language Service APIs, which would require a different strategy implementation.