ternjs.net

Tern Reference Manual

43-55 minutes

Tern consists of several components. Depending on what you are trying to do with it, you will be interested in a different layer. At the very top are the editor plugins. These talk to a Tern server, which is implemented on top of the server module, which uses the inference engine to do the actual type inference.

Contents

The Tern server

The `bin/tern` *binary* (node.js script, really), is used to start a Tern server. You will usually want to let an editor plugin start it for you, but it can be started manually, which can be useful for debugging.

(Note that the basic structure of the server is also available through a programming interface, and that some project, especially those running client-side in a browser, will want to use that instead of the HTTP server described here.)

When started, the server will look for a `.tern-project` file in the current directory or one of the directories above that, and use it for its configuration. If no project file is found, it'll fall back to a default configuration. You can change this default configuration by putting a `.tern-config` file, with the same format as `.tern-project`, in your home directory.

A server write the port it is listening on (which is random) to standard output on startup. It will serve a simple JSON protocol via HTTP on that port. Clients can upload code and ask questions about the code through this protocol.

The following command-line flags are supported:

`--port <number>`
: Specify a port to listen on, instead of the default behavior of letting the OS pick a random unused port.

`--host <host>`
: Specify a host to listen on. Defaults to 127.0.0.1. Pass `null` or `any` to listen on all available hosts.

`--persistent`
: By default, the server will shut itself down after five minutes of inactivity. Pass it a this option to disable auto-shutdown.

`--ignore-stdin`
: By default, the server will close when its standard input stream is closed. Pass this flag to disable that behavior.

`--verbose`
: Will cause the server to spit out information about the

requests and responses that it handles, and any errors that are raised. Useful for debugging.

`--no-port-file`
: The server won't write a `.tern-port` file. Can be used if the port files are a problem for you. Will prevent other clients from finding the server (and may thus result in multiple servers for the same project).

JSON protocol

Sending queries to a Tern server is done by making `POST` requests to the server's port (using `/` as the requests's path) with a JSON document in the body of the request.

This document should be an object, with three optional fields, `query`, `files`, and `timeout`.

The first (`query`) describes the kind of information you are requesting. It may be omitted if the request is only used to push new code to the server (in which case you'll get an empty object, `{}` as response). The `files` property, if given, contains an array of file specifications, as described below. It may be omitted when the query should operate on the code that the server already has, without adding anything new. When the `timeout` field is set, it should contain a number, which is interpreted as the maximum amount of milliseconds to work (CPU work, ignoring I/O) on this request before returning with a timeout error.

A query is an object with at least a `type` property, which determines what kind of query it is. Depending on the type, other properties must or may be present in order to provide further details.

These are the queries that a Tern server understands by default. (Plug-ins may add custom query types.)

`completions`

Asks the server for a set of completions at the given point.

Accepted fields are:

`file`, `end` (required)
: Specify the location to complete at. See below.

`types` (optional, default `false`)
: Whether to include the types of the completions in the result data.

`depths` (optional, default `false`)
: Whether to include the distance (in scopes for variables, in prototypes for properties) between the completions and the origin position in the result data.

`docs`, `urls`, `origins` (optional, default `false`)
: Whether to include documentation strings, urls, and origin files (if found) in the result data.

`filter` (optional, default `true`)
: When on, only completions that match the current word at the given point will be returned. Turn this off to get all results, so that you can filter on the client

side.

`caseInsensitive` (optional, default `false`)
: Whether to use a case-insensitive compare between the current word and potential completions.

`guess` (optional, default `true`)
: When completing a property and no completions are found, Tern will use some heuristics to try and return some properties anyway. Set this to `false` to turn that off.

`sort` (optional, default `true`)
: Determines whether the result set will be sorted.

`expandWordForward` (optional, default `true`)
: When disabled, only the text *before* the given position is considered part of the word. When enabled (the default), the whole variable name that the cursor is on will be included.

`omitObjectPrototype` (optional, default `true`)
: Whether to ignore the properties of `Object.prototype` unless they have been spelled out by at least to characters.

`includeKeywords` (optional, default `false`)
: Whether to include JavaScript keywords when completing something that is not a property.

`inLiteral` (optional, default `true`)
: If completions should be returned when inside a

literal.

The result returned will be an object with `start` and `end` properties, which give the start and end offsets of the word that was completed, an `isProperty` property that holds a boolean indicating whether the completion is for a property or a variable, and a `completions` property holding an array of completions. When one of the `types`, `depths`, `docs`, `urls`, or `origins` options was passed, the array will hold objects with a `name` property (the completion text), and, depending on the options, `type`, `depth`, `doc`, `url`, and `origin` properties. When none of these options are enabled, the result array will hold plain strings.

`type`

Query the type of something.

`file`, `end` (required), `start` (optional)

Specify the expression we want the type of. See below.

`preferFunction` (optional, default `false`)

Set to `true` when you are interested in a function type. This will cause function types to win when something has multiple types.

`depth` (optional, default `0`)

Determines how deep the type string must be expanded. Nested objects will only display property types up to this depth, and be represented by their type name or a representation showing only property names below it.

The returned object will have the following properties:

`type` (string)
: A description of the type of the value. May be `"?"` when no type was found.

`guess` (bool)
: Whether the given type was guessed, or should be considered reliable.

`name` (string, optional)
: The name associated with the type.

`exprName` (string, optional)
: When the inspected expression was an identifier or a property access, this will hold the name of the variable or property.

`doc`, `url`, `origin` (strings, optional)
: If the type had documentation and origin information associated with it, these will also be returned.

`definition`
: Asks for the definition of something. This will try, for a variable or property, to return the point at which it was defined. If that fails, or the chosen expression is not an identifier or property reference, it will try to return the definition site of the type the expression has. If no type is found, or the type is not an object or function (other types

don't store their definition site), it will fail to return useful information.

Only takes `file`, `end` (required), and `start` (optional) fields to specify the expression you want the definition of. See below.

The returned object will have the following properties:

`start`, `end` (offsets, optional)
: The start and end positions of the definition.

`file` (string, optional)
: The file in which the definition was defined.

`context` (string, optional), `contextOffset` (number, optional)
: A slice of the code in front of the definition, and the offset from the start of the context to the actual definition. Can be used to find a definition's location in a modified file.

`doc`, `url`, `origin` (strings, optional)
: If the definition had documentation or an origin associated with it, it will be returned.

`documentation`

Get the documentation string and URL for a given expression, if any.

Takes `file`, `end` (required), and `start` (optional) fields to specify the expression we are interested in. See below.

Returns an object with the following properties:

`doc`, `url`, `origin` (strings, optional)
: The documentation string, url, and the origin of the definition or value, if any.

`refs`

Used to find all references to a given variable or property.

Takes `file`, `end` (required), and `start` (optional) fields to specify the expression we are interested in. See below.

Returns an object with a `name` property holding the name of the variable or property, a `refs` property holding an array of `{file, start, end}` objects, and, for variables, a `type` property holding either `"global"` or `"local"`.

`rename`

Rename a variable in a scope-aware way.

Takes `file`, `end` (required), and `start` (optional) fields to specify the variable we want to rename (see below), and a `newName` property that gives the new name of the variable.

Returns an object whose `changes` property holds an array of `{file, start, end, text}` objects, which give the changes that must be performed to apply the rename. The client is responsible for doing the actual modification.

`properties`

Get a list of all known object property names (for *any* object).

`prefix` (string, optional)
: Causes the server to only return properties that start with the given string.

`sort` (optional, default `true`)
: Whether the result should be sorted.

The returned object will have a `completions` property holding an array of strings, which are the property names.

`files`

Get the files that the server currently holds in its set of analyzed files.

Does not take any parameters. Returns an object with a `files` property holding an array of strings (the file names).

When specifying a location, which is needed for most of the query types, the required `file` field may hold either a filename, or a string in the form `"#N"`, where `N` should be an integer referring to one of the files included in the request (more on that later). The required `end` field is an offset into this file, either a number or a `{line, ch}` object. It should point at the end of the expression the request is interested in, or somewhere inside of it if that doesn't end up pointing inside a sub-expression (in which case the inner expression would be used). An optional `start` field can be used to disambiguate between expressions—if given, the innermost expression that spans the range between start and end will be used.

Offsets into a file can be either (zero-based) integers, or `{line,
ch}` objects, where both `line` and `ch` are zero-based integers. Offsets returned by the server will be integers, unless the `lineCharPositions` field in the request was set to `true`, in which case they will be `{line, ch}` objects.

The format of the `doc` field in responses will, by default, only contain the first few sentences of the documentation string, and have newlines stripped. To get the full string, with newlines intact, you can add a `docFormat` field to your query with the value `"full"`.

Requests that take an input expression also accept a `variable` field which, when given, will cause the server to not look for an expression in the code, but to make up a variable expression with the given name. This does not remove the need to pass in an `end`, since that will be used to determine the scope in which the variable is interpreted.

The `files` property of a request must, if present, hold an array of file descriptions. These can be complete files, which have a `type` property holding `"full"` and `name` and `text` properties holding strings. Complete files will be stored by the server and can be reused in later requests.

Alternatively, you can pass in partial files. This is useful when needing to perform a query on a large file without re-uploading and re-analyzing the whole file. A partial file has a `type` property holding the string `"part"`, a `text` property holding some slice of the file (starting at a line boundary), a `name` property referring to an existing file, and an `offset` property, either an integer or a `{line, ch}` object, indicating the approximate position of the fragment in the file.

To remove a file from the server’s model of the project, you can include an object with a `name` property that identifies the file and a `type` property of `"delete"`.

Programming interface

The base server (without HTTP or configuration-file reading) is implemented in `lib/tern.js`. That package exposes a `Server` constructor that can be used to create a server. It takes an object holding configuration options as argument. These are recognized (all have a default):

`defs` (array of strings)
: The definition objects to load into the server’s environment.

`plugins` (object)
: Specifies the set of plugins that the server should load. The property names of the object name the plugins, and their values hold options that will be passed to them.

`ecmaVersion` (number)

The ECMAScript version to parse. Should be either 5 or 6. Default is 6.

`getFile` (function)

Provides a way for the server to try and fetch the content of files. Depending on the `async` option, this is either a function that takes a filename and returns a string (when not `async`), or a function that takes a filename and a callback, and calls the callback with an optional error as the first argument, and the content string (if no error) as the second.

`async` (bool)

Indicates whether `getFile` is asynchronous. Default is `false`.

`fetchTimeout` (number)

Indicates the maximum amount of milliseconds to wait for an asynchronous `getFile` before giving up on it. Defaults to `1000`.

A server object has the following methods:

`addFile(name: string, text?: string, parent?: string)`

Register a file with the server. Note that files can also be included in requests. When using this to automatically load a dependency, specify the name of the file (as Tern knows it) as the third argument. That way, the file is counted towards the dependency budget of the root of its dependency graph.

`delFile(name: string)`

Unregister a file.

`request(doc: object, callback: fn(error, response))`

Perform a request. `doc` is a (parsed) JSON document as described in the protocol documentation. The `callback` function will be called when the request completes. If an error occurred, it will be passed as a first argument. Otherwise, the resonse (parsed) JSON object will be passed as second argument.

When the server hasn't been configured to be asynchronous, the callback will be called before `request` returns.

`flush(callback: fn())`

Forces all files to be fetched an analyzed, and then calls the callback function.

`on(eventType: string, handler: fn())`

Register an event handler for the named type of event.

`off(eventType: string, handler: fn())`

Unregister an event handler.

`addDefs(defs: object, atFront?: bool)`

Add a set of type definitions to the server. If `atFront` is true, they will be added before all other existing definitions. Otherwise, they are added at the back.

`deleteDefs(name: string)`

Delete a set of type definitions from the server, by providing the name, taken from `defs[!name]` property from the definitions. If that property is not available in the current type definitions, it can't be removed.

`loadPlugin(name: string, options?: object)`

Load a server plugin (or don't do anything, if the plugin is already loaded).

The server fires the following type of events (mostly useful for plugins):

`"reset" ()`

When the server throws away its current analysis data and starts a fresh run.

`"beforeLoad" (file)`

Before analyzing a file. `file` is an object holding `{name, text, scope}` properties.

`"afterLoad" (file)`

After analyzing a file.

`"preParse" (text, options)`

Will be run right before a file is parsed, and passed the given text and options. If a handler returns a new text value, the origin text will be overriden. This is useful for instance when a plugin is able to extract JavaScript content from an HTML file.

`"postParse" (ast, text)`
: Run right after a file is parsed, and passed the parse tree and the parsed file as arguments.

`"preInfer" (ast, scope)`
: Run right before the type inference pass, passing the syntax tree and a scope object.

`"postInfer" (ast, scope)`
: Run after the type inference pass.

`"typeAt" (file, end, expr, type)`
: Run after Tern attempts to find the type at the position `end` in the given file. A handler may return either the given type (already calculated by Tern and earlier `"typeAt"` passes) or an alternate type to be used instead. This is useful when a plugin can provide a more helpful type than Tern (e.g. within comments).

`"completion" (file, query)`
: Run at the start of a completion query. May return a valid completion result to replace the default completion algorithm.

JSON type definitions

To be able to specify the types of things without actually analyzing source code, either because there is no JavaScript source code (as for the built-in types) or because the source code is too big, or because Tern is unable to construct the correct types from the source code, Tern defines a JSON data format for specifying types. A few examples of files in this format can be found in the `defs/` directory in the distribution.

A type definition data structure is basically a tree of objects, where the top-level object specifies variables in the global scope, and the nested objects specify properties of object types. Properties prefixed with an exclamation point (`!`) hold special directives, all other properties refer to variable or property names.

Here is an example:

```
{
  "!name": "mylibrary",
  "!define": {
    "point": {
      "x": "number",
      "y": "number"
    }
  },
  "MyConstructor": {
    "!type": "fn(arg: string)",
    "staticFunction": "fn() -> bool",
    "prototype": {
      "property": "[number]",
      "clone": "fn() -> +MyConstructor",
      "getPoint": "fn(i: number) -> point"
    }
  },
  "someOtherGlobal": "string"
}
```

This defines a library that sets two globals, `MyConstructor` holding a constructor function, and `someOtherGlobal` holding a string. The origin of the types, variables, and properties defined by this document will be `"mylibrary"`, as set by the `!name` property.

The value of a variable or property can be either a string or an object. Strings can be one of the built-in types (`"number"`, `"string"`, `"bool"`), a function type (`"fn(arg1: type1, arg2: type2) -> rettype"`, where `-> rettype` is optional), or an array type (`"[type]"`). Strings can also name types, either by describing the path to the type in the global scope (`"Date.prototype"`) or by referring to one of the local definitions in the `!define` property of the document. Finally, a type can be prefixed with a + to indicate an instance of a constructor (`+Date`).

Objects describe types by enumerating their properties. By default, their type will simply be an instance of `Object`, but a `!type` property can be used to make the type a function or array type, as the example does for `MyConstructor`. Alternatively, a `!proto` property can be used to give the object a custom prototype, for example `"!proto": Element.prototype`.

Documentation can be attached to a type using `!doc`, which should hold a (short) documentation string and `!url`, which should hold a URL that has the full documentation of the type or function being defined.

Finally, a function can be annotated with effects that it has. These aren't currently documented, but you can search for `!effects` in `defs/ecmascript.json` to see some examples. Similarly, a function type string's return type may contain the variables `!0` (the first argument's type, `!N` for the N-1th), `!this` (the self type of the call), and a special property `!ret` (the return type of a function).

Server plugins

Plugins are JavaScript programs that add extra functionality to a server. The distribution currently comes plugins for parsing doc comments, and plugins for node.js and RequireJS, which teach the Tern server about the dependency management mechanisms defined by those systems (as well as, for node.js, including types for the built-in libraries).

A plugin can use several hooks to add custom behavior.

`infer.registerFunction(name: string, f: fn(self, args, argnodes))`

This is a function in the inference engine module that allows custom ways to compute function return types or effects. When a function is specified to return `!custom:myname` or has an effect `"custom myname"`, a call to the function will cause the function registered under `"myname"` to be called with the argument types and argument AST nodes given to the call. This is used, for example, to make calls to `require` trigger the necessary machinations to fetch a dependency and return its type.

`tern.registerPlugin(name: string, fn(Server, options))`

This can be used to register an initialization function for the plugin with the given name. A Tern server, when configured to load this plugin, will call this initialization function, passing in the server instance and the options specified for the plugin (if any). This is the place where you register event handlers on the server, add type definitions, load other plugins as dependencies, and/or initialize the plugin's state.

See the server's list of events for ways to wire up plugin behavior.

`tern.defineQueryType(name: string, desc: object)`

Defines a new type of query with the server. The `desc` object is a property describing the request. It should at least have a `run` property, which holds a function `fn(Server, query)` that will be called to handle queries with a `type` property that matches the given name. It may also have a `takesFile` property which, if true, will cause the server to try and resolve the file on which the query operates (from its `file` property) and pass that (a `{name, text, scope, ast}` object) as a third argument to the `run` function. You will probably need to use the inference module's API to do someting useful in this function.

Doc comments plugin

This plugin, which is enabled by default in the `bin/tern` server, parses comments before function declarations, variable declarations, and object properties. It will look for JSDoc-style type declarations, and try to parse them and add them to the inferred types, and it will treat the first sentence of comment text as the docstring for the defined variable or property.

To turn this plugin off, set `doc_comment: null` in your plugin option.

The plugin understands the following configuration parameters:

`strong`
: When enabled, types specified in comments take precedence over inferred types.

String completion plugin

When enabled, this plugin will gather (short) strings in your code, and completing when inside a string will try to complete to previously seen strings. Takes a single option, `maxLength`, which controls the maximum length of string values to gather, and defaults to 15.

CommonJS module plugin

This plugin implements CommonJS-style (`require("foo")`) modules. It will wrap files in a file-local scope, and bind `require`, `module`, and `exports` in this scope. Does not implement a module resolution strategy (see for example the `node_resolve` plugin). Depends on the `modules` plugin.

Node.js plugin

The node.js plugin, called `"node"`, provides variables that are part of the node environment, such as `process` and `__dirname`, and loads the `commonjs` and `node_resolve` plugins to allow node-style module loading. It defines types for the built-in modules that node.js provides (`"fs"`, `"http"`, etc).

Node.js resolve plugin

This plugin defines the node.js module resolution strategy —things like defaulting to `index.js` when requiring a directory and searching `node_modules` directories. It depends on the `modules` plugin. Note that this plugin only does something meaningful when the Tern server is running on node.js itself.

Modules plugin

This is a supporting plugin to act as a dependency for other module-loading and module-resolving plugins. It understands the following configuration parameters:

`dontLoad`
: Can be set to `true` to disable dynamic loading of required modules entirely, or to a regular expression to disable loading of files that match the expression.

`load`
: If `dontLoad` isn't given, this setting is checked. If it is a regular expression, the plugin will only load files that match the expression.

`modules`
: Can be used to assign JSON type definitions to certain modules, so that those are loaded instead of the source code itself. If given, should be an object mapping module names to either JSON objects defining the types in the module, or a string referring to a file name (relative to the project directory) that contains the JSON data.

ES modules plugin

This plugin (`es_modules`) builds on top of the `modules` plugin to support ECMAScript 6's `import` and `export` based module inclusion.

Webpack plugin

This plugin (`"webpack"`) make use of `enhance-resolve` module from webpack https://webpack.github.io/, so it can understand the `resolve` field of `webpack.config.js` for file resolving. `commonjs` plugin and `es_modules` plugin plugin are loaded in the meanwhile for correct file resolve, this also means you can still have `commonjs` and/or `es_modules` in your `.tern-project` file, but not necessary.

You can use `configPath` option for resolve the config file of webpack like:

```
{
  "libs": [
  ],
  "plugins": {
    "webpack": {
      "configPath": "./lib/webpack.prod.js",
    }
  }
}
```

`configPath` should be a file path relative to `.tern-project`, you can omit it if they're in the same folder.

RequireJS plugin

This plugin (`"requirejs"`) teaches the server to understand RequireJS-style dependency management. It defines the global functions `define` and `requirejs`, and will do its best to resolve dependencies and give them their proper types.

These options are understood:

`baseURL`
: The base path to prefix to dependency filenames.

`paths`
: An object mapping filename prefixes to specific paths. For example `{"acorn": "lib/acorn/"}`.

`override`
: An object that can be used to override some dependency names to refer to predetermined types. The value associated with a name can be a string starting with the character =, in which case the part after the = will be interpreted as a global variable (or dot-separated path) that contains the proper type. If it is a string not starting with =, it is interpreted as the path to the file that contains the code for the module. If it is an object, it is interpreted as JSON type definition.

Angular.js plugin

Adds the `angular` object to the top-level environment, and tries to wire up some of the bizarre dependency management scheme from this library, so that dependency injections get the right types. Enabled with the name `"angular"`.

Third-party plugins

It is possible to write third-party plugins or JSON type definitions and distribute them independently, either as packages using npm or as raw JavaScript files.

When a *name* attribute is specified in the `libs` section of the project configuration, Tern first searches for a file `name.json` in its distribution `defs` directory, then for a file `name.json` in the user's project directory, and finally for an installed package named `"tern-name"` to load as a JSON Type Definition. A JSON Type Definition distributed as an npm package must have a package name matching the pattern `"tern-name"`.

When a *name* attribute is specified in the `plugins` section of the project configuration, Tern first searches for a file `name.js` in its distribution `plugin` directory, then for a file `name.js` in the user's project directory, and finally for an installed package named `"tern-name"` to load as a plugin. A plugin distributed as an npm package must have a package name matching the pattern `"tern-name"`.

The main module for a third-party plugin that will be installed outside of the Tern distribution should export a single method `initialize(ternDir: string)` which takes the location of the Tern distribution loading the plugin as an argument. Tern will call this method immediately after loading the plugin to tell the plugin where to find Tern modules. A plugin with this method will work regardless of its install location relative to the Tern distribution, but plugins without it may fail to find Tern modules or even silently load the wrong Tern modules depending on install location.

Plugin packages can specify the Tern version required as a peer dependency.

Project configuration

A `.tern-project` file is a JSON file in a format like this:

```
{
  "libs": [
    "browser",
    "jquery"
  ],
  "loadEagerly": [
    "importantfile.js"
  ],
  "plugins": {
    "requirejs": {
      "baseURL": "./",
      "paths": {}
    }
  }
}
```

The `libs` property refers to the JSON type descriptions that should be loaded into the environment for this project. See the `defs/` directory for examples. The strings given here will be suffixed with `.json`, and searched for first in the project's own dir, and then in the `defs/` directory of the Tern distribution.

By default, local files are loaded into the Tern server when queries are run on them in the editor. `loadEagerly` allows you to force some files to always be loaded, it may be an array of filenames or glob patterns (i.e. `foo/bar/*.js`). The `dontLoad` option can be used to prevent Tern from loading certain files. It also takes an array of file names or glob patterns.

The `plugins` field may hold object used to load and configure Tern plugins. The names of the properties refer to files that implement plugins, either in the project dir or under `plugin/` in the Tern directory. Their values are configuration objects that will be passed to the plugins. You can leave them at `{}` when you don't need to pass any options.

You can specify an `ecmaVersion` field to configure the version of ECMAScript that Tern parses. The default is 6, and leaving it at that should be safe even for ECMAScript 5 code, but you can set it to 5 as well.

To configure the amount of work Tern is prepared to do to load a single dependency, the `dependencyBudget` option can be added to a project file. It indicates the maximum size of the files loaded in response to a single dependency (through plugins that load dependencies, such as the node and RequireJS plugins), counted in expressions. The default value is 20 000. Files loaded as dependencies of dependencies count towards the budget of the original dependency.

Utilities

`bin/test`
: Runs Tern's own testsuite. Tests are defined in the `test/` directory, as code interspersed with comments that indicate the types and conditions to check for.

`bin/condense`
: Utility for condensing source code down to a JSON type definition file. Takes a list of files to condense, optionally interleaved with filenames prefixed with a `+`, which will be loaded (to provide type information) but not included in the output.

  Pass `--plugin name` or `--plugin name={jsonconfig}` to load plugins. Use `--def file` to load JSON definitions.

Inference engine

The inference engine module (`lib/infer.js`) implements a system that, given a context and an abstract syntax tree (parsed representation of the code), tries to infer the types of the variables and properties in the code.

The parser is implemented in a separate module, Acorn, which provides a regular JavaScript parser, an error-tolerant parser, and a number of utilities for iterating through and searching in abstract syntax trees. The syntax tree format used by Acorn and Tern is described in the Mozilla parser API document (though support for features that are not in ECMAScript 5, such as `let`, is omitted).

A high-level description of the way Tern's type inference algorithm works can be found in a blog post I wrote on the subject.

This module also exposes some utility functions that are useful to implement the services the server exposes.

Context

A context is an object that holds a global JavaScript scope, as well as some meta-information and state used by the type inference process. Almost all operations in the inference module require a context.

To prevent having to pass the context around through every function, a form of dynamic binding is used—the `withContext` function executes a function body with a given object used as the current context.

`infer.Context(defs: [object])`

A constructor function for contexts. `defs` should be an array of type definition objects, which will be used to initialize the global scope.

`infer.withContext(context: Context, f: fn())`

Calls `f` with the current context bound to `context`. Basically, all code that does something with the inference engine should be wrapped in such a call.

`infer.cx() → Context`

Returns the current context object.

`context.topScope`

The top-level scope of the context.

Analysis

To push code into a context, you first parse it, yielding a syntax tree, and then tell Tern to analyze that.

`infer.parse(text: string, options?: {}) → AST`

Parse a piece of code for use by Tern. Will automatically fall back to the error-tolerant parser if the regular parser can't parse the code.

`infer.analyze(ast: AST, name: string, scope?: Scope)`

Analyze a syntax tree. `name` will be used to set the origin of types, properties, and variables produced by this code. The optional `scope` argument can be used to specify a scope in which the code should be analyzed. It will default to the top-level scope.

The same code, or slightly modified variants of the same code, can be analyzed in a context multiple times. The variables and properties that the context knows, as well as the types it assigns to them, will become the union of the variables and types created by the various forms of the code. When incrementally re-analyzing code as it is being edited, this leads to a degradation in the preciseness of the results. To prevent that, it is possible to *purge* all elements that come from a specific origin from a context.

`infer.purgeTypes(origins: [string], start?: number, end?: number)`

Purges the types that have one of the origins given from the context. `start` and `end` can be given to only purge types that occurred in the source code between those offsets. This is not entirely precise—the state of the context won't be back where it was before the file was analyzed—but it prevents most of the noticeable inaccuracies that re-analysis tends to produce.

`infer.markVariablesDefinedBy(scope: Scope, origins: [string], start?: number, end?: number)`

Cleaning up variables is slightly trickier than cleaning up types. This does a first pass over the given scope, and marks variables defined by the given origins. This is indended to be followed by a call to `analyze` and then a call to `purgeMarkedVariables`.

`infer.purgeMarkedVariables`

Purges variables that were marked by a call to `markVariablesDefinedBy` and not re-defined in the meantime.

Types

Tern has a more or less complete implementation of the JavaScript type system.

`infer.Obj(proto, name?: string)`

Constructor for the type that represents JavaScript objects. `proto` may be another object, or `true` as a short-hand for `Object.prototype`, or `null` for prototype-less objects.

`infer.Fn(name: string?, self: AVal, args: [AVal], argNames: [string], retval: AVal)`

Constructor for the type that implements functions. Inherits from Obj. The `AVal` types are used to track the input and output types of the function.

`infer.Arr(contentType: AVal)`

Constructor that creates an array type with the given content type.

`context.num`

The primitive number type.

`context.str`

The primitive string type.

`context.bool`
: The primitive boolean type.

Types expose the following interface (plus some `AVal`-compatibility methods, see below).

`type.name: string`
: The name of the type, if any.

`type.origin: string`
: The origin file of the type.

`type.originNode: AST`
: The syntax node that defined the type. Only present for object and function types, and even for those it may be missing (if the type was created by a type definition file, or synthesized in some other way).

`type.toString(maxDepth: number) → string`
: Return a string that describes the type. `maxDepth` indicates the depth to which inner types should be shown.

`type.getProp(prop: string) → AVal`
: Get an `AVal` that represents the named property of this type.

`type.forAllProps(f: fn(prop: string, val: AVal, local: bool))`
: Call the given function for all properties of the object, including properties that are added in the future.

Object types have a few extra methods and properties.

`obj.proto`
: The prototype of the object, or `null`.

`obj.props`
: An object mapping the object's known properties to AVals. Don't manipulate this directly (ever), only use it if you have to iterate over the properties.

`obj.hasProp(prop: string) → AVal?`
: Looks up the AVal associated with the given property, or returns `null` if it doesn't exist.

`obj.defProp(prop: string, originNode?: AST) → AVal`
: Looks up the given property, or defines it if it did not yet exist (in which case it will be associated with the given AST node).

Abstract values

Abstract values are objects used to represent sets of types. Each variable and property has an abstract value associated with it, but they are also used for other purposes, such as tracking the return type of a function, or building up the type for some kinds of expressions.

In a cleanly typed program where each thing has only a single type, abstract values will all have one type associated with them. When, for example, a variable can hold two different types of values, the associated abstract value will hold both these types. In some cases, no type can be assigned to something at all, in which case the abstract value remains empty.

Abstract values expose the following interface:

`infer.AVal()`
: Constructor. Creates an empty AVal.

`aval.addType(type: Type, weight?: number)`
: Add a type to this abstract value. If the type is already in there, this is a no-op. `weight` can be given to give this type a non-default weight, which is mostly useful when adding a provisionary type that should be overridden later if a real type is found. The default weight is `100`, and passing a weight lower than that will make the type assignment “weak”.

`aval.propagate(target: Constraint)`
: Sets this AVal to propagate all types it receives to the given constraint. This is the mechanism by which types are propagated through the type graph.

`aval.hasType(type: Type) → bool`
: Queries whether the AVal *currently* holds the given type.

`aval.isEmpty() → bool`
: Queries whether the AVal is empty.

`aval.getType(guess?: bool) → Type?`
: Asks the abstract value for its current type. May return `null` when there is no type, or conflicting types are present. When `guess` is `true` or not given, an empty

AVal will try to use heuristics based on its propagation edges to guess a type.

`aval.getFunctionType() → Type?`
Asks the AVal if it contains a function type. Useful when you aren't interested in other kinds of types.

Abstract values that are used to represent variables or properties will have, when possible, an `originNode` property pointing to an AST node.

As a memory-consuming hack, types also expose the interface of abstact values, so that in cases where an AVal is expected, but the precise type is known, the type object can simply be used.

The `infer.ANull` value is a special AVal-like object that never holds any types, and discards types added to it. It can be used as a placeholder in situations where we either aren't interested in the types, or there simply are no types.

Constraints

Constraints are things that can receive values. They use the same `addType` method to receive them, which causes all AVals to also be useable as constraints. The inference engine defines a number of additional constraints to propagate values in more indirect ways. See the blog post on the inference algorithm for some examples.

`infer.constraint(methods: object) →`

`constructor`

This is a constructor-constructor for constraints. It'll create a constructor with all the given methods copied into its prototype, which will run its `construct` method on its arguments when instantiated.

Beyond `addType`, there are a few optional methods that constraints can expose to tell the system something about itself.

`constraint.typeHint() → Type?`

May return a type that `getType` can use to "guess" its type based on the fact that it propagates to this constraint.

`constraint.propHint() → string?`

May return a string when this constraint is indicative of the presence of a specific property in the source AVal.

Scopes

Scopes are derived from the `Obj` type, and variables are represented the same way as properties.

`infer.Scope(parent?: Scope)`

Constructor for scope objects. The top scope won't have a parent.

`scope.defVar(name: string, originNode: AST) → AVal`

Ensures that this scope or some scope above it has a property by the given name (defining it in the top scope if it is missing), and, if the property doesn't already have an `originNode`, assigns the given node to it.

Utilities

These are miscellaneous utilities that come in helpful when doing code analysis.

`infer.findExpressionAt(ast: AST, start: number?, end: number, scope?: Scope) → {node, state}`

Searches the given syntax tree for an expression that ends at the given end offset and, if `start` is given, starts at the given start offset. `scope` can be given to override the outer scope, which defaults to the context's top scope. Will return a `{node, state}` object if successful, where `node` is AST node, and `state` is the scope at that point. Returns `null` if unsuccessful.

`infer.findExpressionAround(ast: AST, start: number?, end: number, scope?: Scope) → {node, state}`

Similar to `findExpressionAround`, except that it will return the innermost expression node that spans the given range, rather than only exact matches.

`infer.expressionType(expr: {node, state}) → AVal`

Determine an expression for the given node and scope (as returned by the functions above). Will return an `AVal` or plain `Type`.

`infer.scopeAt(ast: AST, pos: number, scope?: Scope) → Scope`

Find the scope at a given position in the syntax tree. The `scope` parameter can be used to override the scope used for code that isn't wrapped in any function.

`infer.findRefs(ast: AST, scope: Scope, name: string, refScope: Scope, f: fn(AST, Scope))`

Will traverse the given syntax tree, using `scope` as the starting scope, looking for references to variable `name` that resolve to scope `refScope`, and call `f` with the node of the reference and its local scope for each of them.

`infer.findPropRefs(ast: AST, scope: Scope, objType: Obj, propName: string, f: fn(AST))`

Analogous to `findRefs`, but used to look for references to a specific property instead. Whereas `findRefs` is precise, this is dependent on type inference, and thus can not be relied on to be precise.

Whenever the code in `lib/infer.js` guesses a type through fuzzy heuristics (through `getType` or `expressionType`), it sets a flag. The following two function allow access to this flag:

`infer.didGuess() → bool`

Test whether the guessing flag is set.

`infer.resetGuessing(val?: bool)`
: Reset the guessing flag.

Editor plugins

If your editor of choice is not yet supported, you are encouraged to try and port one of the existing plugins to it. When figuring out how things work, the code powering the demo, in `doc/demo/demo.js`, might also come in useful.

All these plugins use the node.js-based server, and thus require that (as well as npm) to be installed.

Emacs

The Emacs mode is part of the main tern repository. It can be installed as follows:

1. Make sure you are using Emacs 24 or later. The Tern mode requires lexical scoping.
2. Clone this repository somewhere. Do `npm install` to get the dependencies.
3. Make Emacs aware of `emacs/tern.el`. For example by adding this to your `.emacs` file:

   ```
   (add-to-list 'load-path "/path/to/tern/emacs
   /")
   (autoload 'tern-mode "tern.el" nil t)
   ```
4. Optionally set `tern-mode` to be automatically enabled for your JavaScript mode of choice. Here’s the snippet for `js-`

`mode`:

```
(add-hook 'js-mode-hook (lambda () (tern-mode t)))
```

The Emacs mode uses the `bin/tern` server, and project configuration is done with a [`.tern-project`] file.

Buffers in `tern-mode` add a `completion-at-point` function that activates Tern's completion. So, unless you rebound the key, `M-tab` (or `C-M-i`) will trigger completion.

When the point is in an argument list, Tern will show argument names and types at the bottom of the screen.

The following additional keys are bound:

`M-.`
: Jump to the definition of the thing under the cursor.

`M-,`
: Brings you back to last place you were when you pressed `M-.`.

`C-c C-r`
: Rename the variable under the cursor.

`C-c C-c`
: Find the type of the thing under the cursor.

`C-c C-d`
: Find docs of the thing under the cursor. Press again to open the associated URL (if any).

Auto-Complete

If you want to use `auto-complete.el` for completion, append following codes:

```
(eval-after-load 'tern
   '(progn
      (require 'tern-auto-complete)
      (tern-ac-setup)))
```

If `tern-ac-on-dot` is non-nil (default), typing `.`(dot) invokes auto-complete to select completions. Calling the command `tern-ac-complete`, one can invoke auto-complete manually.

Vim

The Vim plugin is maintained in a separate repository. Please see its README for details.

Sublime Text

There are two implementations of Sublime Text modules for Tern. One written by me, at https://github.com/ternjs/tern_for_sublime. This one uses the node.js-based server. The other is written by Sergey Chikuyonok, and can be found at https://github.com/emmetio/sublime-tern. It uses a Python V8 bridge to run the server.

See the readme files in those repositories for details.

Eclipse / Java

tern.java is a Tern client written in Java. It includes Eclipse integration.

gedit

tern_for_gedit integrates Tern with the gedit editor.

TextMate

JavaScript Tern Completion.tmbundle integrates Tern with TextMate.

Related Software

This is a place to gather links to software that is built on Tern. Drop me an email if you want to suggest a link.

jsctags

jsctags is a tool that generates ctags files from JavaScript code by using Tern's inference engine.

CodeMirror Tern addon

The CodeMirror browser-based editor has an addon for integrating with Tern.

tern.ace

tern.ace integrates Tern in the ACE editor.

tern.orion

tern.orion integrates Tern in the Orion editor.