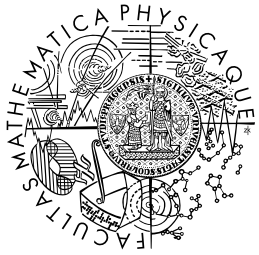

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Ondřej Roztočil

Type providers for TypeScript

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Tomáš Petříček, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank Kateřina Průšová for going through this academic ordeal with me, even making it fun at times. I would also like to thank my advisor Tomáš Petříček for the help and insight he provided me and for showing me that it does not have to be an ordeal after all.

Title: Type providers for TypeScript

Author: Ondřej Roztočil

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: Type systems of programming languages generally do not understand external data. In statically typed programming languages, developers handle data by writing typed interfaces and data access code by hand or using external code generation tools. Type providers, originally developed for F#, are an alternative, more integrated approach to code generation that improves compile-time safety and developer experience by inferring types from data samples.
This thesis explores the implementation of type providers in TypeScript. We propose a design for the feature and implement a functional prototype as an extension of the current TypeScript compiler. We demonstrate the feature's usefulness by implementing type provider packages for CSV, XML and JSON.

Keywords: type providers, TypeScript, data access, code generation

Název práce: Type providers pro TypeScript

Autor: Ondřej Roztočil

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Typové systémy programovacích jazyků obecně nerozumí externím datům. Ve staticky typovaných programovacích jazycích vývojáři pro práci s daty vytvářejí typy a přístupový kód buď ručně nebo s použitím externích nástrojů pro generování kódu. Mechanismus type providers, původně vyvinutý pro jazyk F#, představuje alternativní přístup ke generování kódu, který zlepšuje jak bezpečnost v době kompilace, tak efektivitu vývojářů, za pomoci odvozování typů ze vzorků dat.
Tato práce zkoumá implementaci type providers v jazyce TypeScript. V rámci práce jsme navrhli řešení a implementovali funkční prototyp ve formě rozšíření současného překladače TypeScriptu. Užitečnost této funkce demonstrujeme implementací type provider balíčků pro CSV, XML a JSON.

Klíčová slova: type providers, TypeScript, přístup k datům, generování kódu

Contents

1. Introduction

Processing data from external data sources is the bread and butter of a large class of real-world applications. Reading and writing data files or structured messages in various formats, querying databases, or interacting with APIs of other services are common activities that all in some form involve mapping between the (implicit or explicit) schema of the external data source and the type system of the programming language.

The importance of this task is indicated by the popularity of libraries and tools such as JSON or XML parsers and validators, object-relational mapping frameworks, database query builders, schema-based code generators, or clients libraries for established HTTP APIs — in many mainstream programming languages.

In most cases, the type systems of programming languages do not "understand" external data. By that, we mean that even if information about the structure of the data, names of entities or fields, types of the associated values, and so on, are available during compilation or while the developer writes the code (henceforth called the design-time), in the form of a data sample or an explicit schema, this information is not automatically used during type checking or when providing feedback in a code editor.

In the context of statically typed programming languages, developers can integrate the external data into the language's type system by supplying type declarations that describe the structure, names, and permissible values — in terms of the language's constructs, such as classes, interfaces, methods, and so on. Such declarations typically need to be accompanied by a (handwritten or generated) set of infrastructural functions that perform the necessary IO operations, parsing, and validation to access the data source and transform the data into the declared representation. In return, the developers get the well-documented benefits of strong static typing [1], in particular, better detection of common errors (mistyped names, wrong value types), better IDE support (auto completions), improved maintainability, and potentially increased performance.

This, however, introduces the problem of providing a correct representation (model) of the data and keeping it accurate when the data schema changes. Broadly speaking, there are two categories of approaches to creating and updating such models: handwritten libraries and code generation based on some external schema. Both lead to different scalability and maintainability challenges. Writing the necessary code by hand can be costly and creates opportunities for human error. Code generation has been criticized [2] [3] for creating a fragile dependency, complicating the build process, interacting badly with source control, or generating potentially bloated code.

With dynamically typed languages like JavaScript, developers are more free to avoid modeling the external data in exchange for losing the benefits of static typing. However, any serious application that wants to maintain at least run-time correctness will necessarily need to perform a comprehensive validation of the external data it reads. In practice, writing such validation code or a schema for a validation library is similar to data modeling for statically typed languages and reintroduces similar trade-offs regarding handwriting and generating code.

1.1 Introducing type providers

An interesting variant of the code generation paradigm is the concept of type providers. A type provider, generally speaking, is a compile-time component for a programming language that, given a data sample or a data schema, infers an appropriate representation of said data in the type system of the language and provides this inferred type information to the language's compiler to be used for type checking or supporting code editor features. Depending on implementation, a type provider can also generate run-time code that can be used for safe and efficient data access.

It has been argued that type providers improve upon some of the issues of traditional handwritten and code generation-based methods of handling external data. They can increase the safety and robustness of the programs, improve the developer experience, and facilitate explorative programming while being code-focused and not requiring the developer to interact with external tools [4]. Schema-based type providers can be seen as a direct replacement of similar external code generation tools. Meanwhile, type providers based on inference from actual data samples present a different approach to the data modeling problem, moving the focus to the engineering of representative samples, which can, in some situations, be more readily available than schemas.

Type providers have been first explored and most practically realized in the F# programming language [4, 5, 6]. The F# type providers are implemented as special libraries that users can add to their projects and invoke in code using standard F# syntax for generic types. When checking such types during compile-time or design-time (in a compatible IDE), the F# compiler executes code contained in the type provider assembly and retrieves the inferred type definitions and code.

A distinguishing feature of the F# type providers is that the type provider does not need to examine the entire data source at once and does not need to generate actual F# (.NET) types. Instead, the type provider can return partial typing information lazily as needed (based on the user's activity). The provided types also can be *erased* during compilation, meaning that the specific types are replaced by a smaller set of run-time compatible types (e.g., strings or basic collections), thus reducing the need for code generation and size of the assembly. Thanks to these features, the F# type providers should be able to scale to work with much larger information spaces (e.g., very large databases or public APIs) than what can be supported with traditional eager code generation [4].

Apart from the F# implementation of type providers as dynamically loaded compiler plugins, there have been other implementations based on different compile-time capabilities of the host languages. In the dependently typed language Idris [7], type providers have been realized using the ability of specially annotated compile-time evaluated functions to return types without the need for code generation. The powerful macro system of Scala 2 enabled a type provider-like mechanism to be implemented without additional modification to the language's capabilities [8]. However, Scala 3 has removed support for the so-called whitebox macros (i.e., macros that can resolve to any type), making the previous approach to implementing type providers unfeasible.[9]

1.2 Type providers for TypeScript

This thesis investigates the possibility of implementing type providers into the TypeScript programming language. TypeScript is a widely used [10] extension of the dynamically-typed language JavaScript, focused on adding gradual static typing and best-effort compile-time safety while being able to be transpiled into plain JavaScript [11].

In recent years, TypeScript has been used extensively in many areas of development, particularly for web, mobile, and desktop GUI applications and backend services. These applications commonly involve manipulating external data or accessing HTTP APIs and databases, all presenting a potential use case for type providers.

Apart from these practical motivations, we argue that type providers are a particularly good fit for TypeScript. Among other reasons:

- Type providers follow TypeScript's design goals and constraints, which focus on adding best-effort static type-checking to JavaScript.
- Due to TypeScript's unsoundness and JavaScript's weak run-time typing, there is an increased need for robust handling of external data.
- Similar to F#, TypeScript has strong type inference capabilities and requires a limited number of type annotations, which makes it well-suited for provided types.
- There is a longstanding interest in this feature in the TypeScript community [12], and even greater demand for introducing a general compiler plugin mechanism ([13, 14]). A lot of the use cases for the latter could be covered in a more controlled way by a type provider mechanism.

Admittedly, TypeScript's stated design principles also include not providing run-time functionality and not emitting different code based on the results of the type system [15]. We argue, however, that these intentional limits apply to the language itself and the behavior of the core compiler. Introducing an extensible plugin mechanism that allows end users to opt-in to a code generation functionality (as they often already do in a less convenient and less type-safe way [16]) is not in conflict with these principles.

1.3 Thesis goals

As we have argued throughout this introduction, type providers are a helpful mechanism, an interesting alternative to traditional methods of dealing with external data in statically typed languages, and a natural fit for TypeScript. However, the current TypeScript compiler is not implemented with a type provider feature in mind and does not support plugin extensibility powerful enough to enable implementing it from outside. Therefore, modification of the compiler is required.

Based on this, the high-level goals of this thesis are as such:

1. To investigate the problem of introducing a type provider-style feature into TypeScript and its current compiler.
2. Propose a design for the feature and its implementation.
3. Implement a functional prototype.
4. Implement type provider packages for CSV, XML, and JSON to demonstrate the solution's viability.

1.4 Thesis structure

The remainder of the thesis is structured into the following chapters.

Chapter 2 contains a preliminary analysis of the problem of adding type providers to TypeScript in the broader context of the JavaScript/TypeScript ecosystem, related work, and general programming language design concerns. We gather a list of requirements for the implementation and present initial design decisions and constraints.

Chapter 3 documents the implemented solution. Section 3.1 provides an overview of the TypeScript compiler and language service and describes the modifications we made to support the type provider mechanism.

Section 3.2 deals with the implementation of type provider packages. We discuss shared concerns for typical providers and present the showcase type providers for CSV, XML, and JSON.

Chapter 4 contains documentation for using the implemented solution. We give instructions for setting up a development environment with the modified compiler and for using the existing type provider packages. The Section also contains a step-by-step tutorial for creating a new type provider package for a simple data format.

Chapter 5 evaluates the outcomes of the thesis and discusses possible future work.

2. Problem analysis

In this chapter, we present a general analysis of the problem of adding type providers to TypeScript. We consider the project in the wider context of the JavaScript (JS) and TypeScript (TS) ecosystems, related work, and general programming language design concerns. We gather a list of requirements for the implementation and present initial design decisions and constraints.

2.1 Case study: Reading data in JavaScript and TypeScript

Before we proceed with the analysis, we want to begin with a motivating example that illustrates common issues of handling external data in JavaScript and TypeScript. We will use a simple, practical scenario involving parsing data from a CSV file.

To begin, let us assume that the programmer expects to parse CSV files like the following:

```
Item,Cost,Paid
"JNLW2",200,true
"X4QTM",150,false
"9RFVH",800,false
"D6K3P",3600,true
```

Using a popular NPM package `papaparse`, we can load the data and parse it into individual values as such:

Code Listing 2.1 Loading untyped CSV data

```
import * as fs from "node:fs";
import * as Csv from "papaparse";

const fileContent = fs.readFileSync("../data.csv", "utf8");
const rawData = Csv.parse(fileContent, { header: true }).data;
```

This gives us the data as an array of rows represented as records with a string field for each column. (Note that to determine that this is the case, we either had to examine the result during run time or read the documentation because the library types the return type as `unknown[]`.)

Now, suppose we want to process the values from the `Cost` column as numbers (e.g., sum them up), and we want our processing code to be statically type-checked.

To do that, we need to do two things:

1. Declare a TypeScript interface that specifies the name and type of fields in your fully parsed data.

2. Write code that transforms the input records with string fields into records that satisfy the declared interface.

Code Listing 2.2 Mapping CSV data to typed representation

```
interface Row { item: string; cost: number; paid: boolean; }

const parseData = (data: Record<string, string>[]) =>
    data.map(row => ({
        item: row["Item"],
        cost: Number(row["Cost"]),
        paid: row["Paid"] === "true"
    }));

const sumCosts = (rows: Row[]) =>
    rows.reduce((sum, row) => sum + row.cost, 0);

const data = parseData(rawData as Record<string, string>[]);
const result = sumCosts(data);
```

This code introduces some potential issues:

1. We needed to write the interface and the mapping code by hand and would need to maintain it in the future. This is trivial for such a simple case, but the amount of work would naturally grow with the complexity of the data.

2. The mapping code in the `parseData` function needs to access fields in the parsed rows using string constants. This can introduce bugs, particularly because access to a non-existing property is not a run-time error in JavaScript and simply returns an ‘undefined‘ value instead.

3. The mapping code performs no validation. If the actual data differs from expectations or schema later changes, many bugs are possible. For example, it is not uncommon to use text to indicate missing values or special statuses in otherwise numeric columns. This would not throw a run-time type error but pollute the data with `NaN` values, most likely leading to run-time safe but logically invalid results in downstream code. The optimistic parsing of the `Paid` value can lead to unexpected consequences if input data includes other representations of boolean values, and so on.

It is certainly possible to write validation code that performs all necessary run-time checks so that the program either gives the expected result or throws an exception, regardless of the shape of the actual input data. However, the point we are trying to illustrate is that when crossing the inherently problematic boundary between code and external data, the correctness of the program hinges on the completeness and correctness of the run time code that handles this conversion.

The issue is even more prominent in TypeScript. Compared to languages such as Java or C#, TypeScript has more sources of unsoundness [17] while also targeting run time environment with more permissive type checks, including implicit type coercions. At the same time, when writing downstream code that uses the type annotated data loaded from an external source, programmers would reasonably expect the TypeScript’s apparent compile-time guarantees to hold and not check the actual values as defensively as they would in a purely dynamic language.

2.2 Type safety

To help deal with the issues illustrated so far, we propose introducing the mechanism of type providers into TypeScript.

Such a mechanism would allow the creation of special libraries for various formats or data sources that a user could opt into using in their TypeScript projects. Then, the user could invoke the type provider by adding certain, yet undetermined, constructs into their code.

When the (appropriately modified) TypeScript compiler encounters this construct during compilation, it loads the type provider code and executes it, passing it relevant user-specified static options (e.g., path to a data sample). This invokes the type inference process implemented by the respective type provider package (e.g., CSV provider). The resulting set of inferred (provided) types is further used by the compiler to type-check the user's source files using regular typing rules.

We can re-state the goal of implementing such a mechanism as the first requirement for our solution:

R1 **The TypeScript compiler will be able to invoke a type provider in the compile-time.**

R2 **Type providers will be able to generate static types based on the user's configuration that the compiler can use for standard type checking.**

As we have argued, static typing is not enough to achieve actual type safety when dealing with external data in TypeScript and JavaScript. Therefore, the provided types must also be accompanied by run-time parsing and validation code that verifies that run-time types of successfully loaded values match the compile-time types, thus keeping the static guarantees the user might rely upon.

R3 **Type providers will be able to generate data access code to ensure that run-time types match the provided static types.**

2.3 New syntax

A basic design question for the type provider solution is how exactly the user invokes the provider mechanism or acquires the provided types to use in their code. There are many factors to consider when adding a new syntax to a programming language or reusing existing syntax for a new feature, including:

- Ambiguity — Does it not introduce ambiguity into the grammar?
- Readability — Is it easy to understand for a user? Is it clearly differentiated from other features?
- Expressiveness — Can the syntax express the information needed for the feature? How well?
- Consistency — How consistent is it with the existing language conventions?

- Backwards compatibility — Is it safe to add without breaking existing code?
- Design space — How much design space does it take from future features?
- Ease of implementation — How difficult is it to support in the context of existing compiler implementation?

Based on these considerations, we propose to extend the existing TypeScript syntax [18] for (static) module imports based on the ECMAScript module syntax [19]. The standard import syntax is shown in code listing 2.3.

Code Listing 2.3 Standard TypeScript import syntax

```
// Import from a package
import { z } from "zod";

// Multiple named imports from relative path
// SomeType is a TypeScript interface, class or type
// The type keyword is optional by default
import { myFunc, type SomeType } from "./my-lib";

// Aliased import
import { z as validator } from "zod";

// Namespace import
// Identifier "fs" will contain all exported object from the "node:fs" module
import * as fs from "node:fs";

// Import with import attribute block
import jsonData from "./data.json" with { type: "json" };
```

We propose to introduce a new modifier keyword `provided` and use the existing import attribute block (after the `with` keyword) so that specifying imports of provided types and values would be done as shown in the code listing 2.4.

Code Listing 2.4 Proposed TypeScript syntax for imports of provided code

```
// Named import of provided type based on file sample
import provided { Row } from "csv-provider" with { sample: "../sample.csv" };

// Namespace import of all provided code
import provided * as BookData from "xml-provider" with { sample; "../books.xml" };
```

We argue that such syntax applies well to the question raised above.

- It is familiar to TypeScript users and aligns with trends in the development of JavaScript. It allows treating the provided code as a normal (if virtual) module.

- The import attribute block (after the `with` keyword) provides flexibility for passing arguments to the type provider.
- The `provided` keyword clearly states intent and differentiates the usage from a regular import. It ensures non-ambiguity of the grammar and preserves both backward compatibility with existing code and design space for future development (particularly for keys added to the import attribute block).
- Supporting the syntax should be easy using functionality already present in the compiler.

R4 Including provided types into the user's code base will be done by using a new `provided import` statement syntax.

To illustrate the proposed type provider solution with the new syntax, let us return to the example of reading CSV data from Section 2.1. Reading the file and performing the same computation as before could be done like this:

Code Listing 2.5 Reading CSV data with a type provider

```
1 import provided { Row, loadFileSync } from "@ts-providers/csv"
     with { sample: "../sample.csv" }
2
3 const data = loadFileSync("../data.csv");
4
5 const sumCosts = (rows: Row[]) =>
6     rows.reduce((sum, row) => sum + row.cost, 0);
7
8 const sum = sumCosts(data);
```

Using the type provider is at least a partial improvement on all of the previously discussed issues:

- Both the `Row` interface and the code that allows reading the CSV data have been generated in the background, based on the shape of data inferred from the `sample.csv` file specified in the provided import statement, thus reducing the amount of code that needs to be written.
- The generated code validates the data it reads so that the run-time types match the static typing of the interface or the read operation throws an error. E.g., unless a column is inferred as nullable, successfully loaded data will never contain a `null` or `undefined` value in the respective field.
- There is no unsafe access to fields using handwritten string constants and guessing types, thus reducing the potential for human error.
- If the sample changes in a way that would invalidate the user's code using the data, static type checking against the newly generated interface causes a compilation error, notifying the user about the change.

2.4 Design-time features

Apart from the increased safety guarantees, a key benefit of static typing is that it enables useful design-time features for code editors, such as auto-completion, live in-code diagnostics (
squiggles), and automated refactorings.

We consider adding new automated refactorings and code fixes for type providers out-of-scope in the prototype implementation. However, existing editor actions that also apply to type provider-related code should either work properly or not result in errors.

R5 Provided types will be compatible with the standard auto-completion and error diagnostic features of the TypeScript language service.

When working with provided types, it might be often useful for the user to be able to inspect the generated code. This is a feature that, to our knowledge, has not been implemented by any of the previous code generation-based implementations of type providers. Among other reasons, it would enable users to simply copy the provided code and tweak it for their purposes or simply make it a permanent part of their code base.

While it might be less confusing for users to implement this as a new IDE action, this would likely be a much more complex task than modifying the behavior of the existing
Go to definition feature for provided types.

R6 Go To Definition feature that enables inspecting the provided code will be supported.

To support these design-time features, TypeScript implements not only the standard command line compiler but also a language service [20]. The language service shares most of the functionality, such as parsing and type checking, with the compiler. However, it is a long-lived program optimized for on-demand answering of type queries instead of efficient batch processing.

TypeScript is distributed with two programs: the command line compiler invoked using the `tsc` command and the `tsserver`. The latter encapsulates both the compiler and the language service and answers queries from other programs, such as code editors, via a JSON API.

Our type provider solution should be compatible with `tsserver` and, in particular, with the official Visual Studio Code TS extension, which uses `tsserver` internally. We selected Visual Studio Code as it is the most popular IDE for TypeScript development [21]. Other editors might work but will not be tested.

R7 The language service features for provided types will be supported in recent stable versions of Visual Studio Code.

2.5 Execution characteristics

We should consider how the interaction between the compiler and a type provider is going to work. Technical specifics of provider invocation in the prototype will be discussed in Section 3.1. Here, we want to specify the goal of our solution.

As described in Section 1.1, the F# implementation of type providers supported on-demand (lazy) provider invocation that returns partial results to satisfy only an immediate type system query. This allowed the implementation of providers that can be used to access very large data with very large schemas (described by the authors as "internet-scale" [5]).

To limit the scope of the prototype implementation and based on a preliminary analysis of the current architecture of the TypeScript compiler, we decided not to pursue this direction. The invocation mechanism between the compiler and a provider will be eagerly executed when the compiler deals with module imports of a currently processed file. The provider will be expected to generate the entire code, even if that would lead to excessively large results.

R8 Type providers will be invoked eagerly by the compiler and generate the entire provided code for the data source at once.

Another question is in what form will the type providers return types and/or code to the compiler? We could create some form of intermediate representation specialized for the provider solution. This could enable some optimizations or standard preprocessing that would be otherwise more difficult to perform.

However, since we decided to invoke providers eagerly and have them generate the entire code for each invocation, it seems practical to let the providers generate some actual representation used by the compiler.

The most feasible solution seems to be to generate abstract syntax tree (AST) nodes which are normally constructed while parsing a TypeScript source file. Types for the TypeScript AST are part of the public compiler API which also contains factory methods for correct creation of different kinds of nodes. There are also third-party libraries that streamline the creation of the TypeScript AST that can be used by the type provider implementors. For example, the ts-morph library [22] exposes a builder API for creating TypeScript code and allows programmatic parsing of the AST from code in string literals.

Furthermore, the AST representation is used throughout the compiler, meaning that if synthesized properly, most of the compiler's subsystems should be able to handle the provided code without significant modifications.

R9 When type provider inference succeeds, a valid code will be returned in the form of abstract syntax tree representation used by the TypeScript compiler.

It can be expected that some type providers would need to perform operations that can be performed in JavaScript only asynchronously. Reading from files can be done in the Node.js environment synchronously (thanks to the custom native implementation of the `readFileSync` function), but making HTTP requests, for instance. does not have any synchronous method.

For this reason, the provider invocation mechanism should support both synchronous and asynchronous providers that return a `Promise` instead of an

immediate value. The compiler needs to be able to wait for the completion of the promise because the rest of the compiler pipeline cannot continue in the background. Considering that the current TypeScript compiler is designed entirely as a synchronous application, there might be significant issues with implementing such support properly without a major rewrite. However, workaround solutions should be investigated for the prototype implementation.

R10 **Both synchronous and asynchronous type providers will be supported.**

2.6 Error handling

It is expected that the compiler and the language service should never crash with an uncaught exception just because it cannot deal with the user's source code. The intended method of error handling in the compiler is by emitting diagnostic messages.

Type providers implemented as third-party compiler plugins create an unpredictable source of run-time exceptions. The provider invocation mechanism needs to be implemented defensively to prevent a faulty provider package from crashing the compiler.

R11 **No compiler crashes will occur due to type provider invocation. In case the type provider crashes, the compiler will return a generic diagnostic instead.**

However, this does not mean that type provider must always succeed. The provider must be able to return diagnostic messages in case of issues such as invalid configuration and unavailable or malformed samples. As the diagnostic message interfaces are already a part of the public compiler API, the providers can create the necessary structures directly and pass them to the compiler, similarly to generating code.

R12 **Type provider will be able to return error diagnostics instead of or in addition to the provided code.**

2.7 Performance

The performance overhead of the general type provider mechanism must not degrade developer experience. Specific type provider packages should be implemented (and *be able* to be implemented) in such a way that provides a reasonable upper bound on the execution time of each type inference operation. When no type provider is used in the code base, there should be no performance regression in the compiler or other parts of the tooling when compared to their versions without type provider support. However, actual performance optimization is not a priority for our prototype implementation.

R13 **Adding support for type providers will not degrade compiler performance on code that does not use type providers.**

R14 **The type provider mechanism itself should not introduce significant performance regression. Ensuring reasonable final performance and responsivity will be the responsibility of individual provider packages.**

2.8 Emitted JavaScript code

In order to execute TypeScript code, it has to be transpiled into JavaScript first.[1] This process is handled in the compiler by the transformer pipeline, which, in a series of steps, removes all TypeScript-only constructs from the code and optionally replaces JavaScript features unsupported by the target version of the ECMAScript standard.

Because we are introducing a new kind of import statement with a new TypeScript-only syntax, we need to add a new transformer step that removes the provided syntax and leaves only a regular TypeScript import statement. The rest of the transpilation should be able to proceed without any modification as the imported values and types are used in the rest of the code as if they were imported from an actual external module.

The user code will typically depend not only on provided static types but also on generated run-time code. Therefore, the provided code also needs to be transpiled into JavaScript and emitted along with the user code. The module specifiers in the import statements for the provided code will need to be rewritten so that they point to the correct emitted source files.

R15 **TypeScript code using type providers will be transpiled into valid JavaScript code. Any provider-specific syntax will be removed in the process.**

R16 **Provided source code will be emitted and imported in the transpiled user code so that provided functions and values can be properly used during run-time.**

Unlike in F#, which targets the .NET runtime, there is no need to implement type erasing as a special feature for type providers [4], as TypeScript types are all erased during compilation by default. However, there is a question if the compiler should try to analyze the static imports and exports in the user code in order to determine which run-time components of the provided code are actually used and not emit unused code. Because the compiler does not perform this operation (also called tree-shaking) on the regular code, we will not try to implement it for the provided code.

In practice, the responsibility for removing unnecessary code can be delegated to external bundlers such as Webpack [23] or Rollup [24]. Bundling in the context of JavaScript means taking the contents of multiple files and merging them into a single file, modifying names, and cross-references in order to preserve imports and

[1] There are multiple runtimes that, from the user's perspective, support executing TypeScript code directly (e.g. Deno or Bun.js). However, these runtimes are not supported for the prototype implementation of type providers, because they internally use their own versions of the TypeScript compiler which we are not able to modify.

exports and avoid conflicts. The compiler also has built-in bundling capabilities. However, as this functionality can be used only for legacy module systems and not with the current ESM and CommonJS systems, it will not be supported by the type provider solution.

2.9 Extensibility

An important feature of type providers is the fact that although they are compile-time components, they are not built into the compiler, and creating new providers does not require modification of the compiler. To achieve this extensibility in TypeScript, we need to follow the F# model of type providers as externally loaded compiler plug-ins because TypeScript does not have compile-time metaprogramming capabilities such as Idris [7] or Scala [8].

Currently, there is support for the TypeScript language service. However, these plugins can only affect a limited number of code editor features, cannot participate in type checking, and have no effect on the command line compiler itself [25].

R17 Creating new type providers will be possible without requiring additional modifications of the compiler or the language service.

The standard way to configure a JavaScript or TypeScript project, including installing and versioning dependencies and managing the build output, is via the `package.json` configuration file. This file is used by package managers such as NPM and module loaders of runtimes such as Node.js. Type provider packages should behave in this regard like regular packages distributed from a public or private package repository. That is, users should be able to add a type provider (or its specific version) to their project by specifying it as a dependency in the `package.json` file or by executing an appropriate package manager command such as `npm install <provider-package-name>`.

There are currently multiple competing package managers in the JS/TS ecosystem, including NPM and multiple NPM-compatible solutions such as Yarn and pnpm. For the purpose of the prototype implementation, we will only aim to support NPM, as it is the most popular package manager and the native solution for Node.js (our target runtime). However, other NPM-compatible managers should work in practice as well.

R18 Type providers will be distributed and installed as regular NPM packages. Execution of type providers will be supported when running the compiler on Node.js.

It can be expected that typical type provider implementations will share a non-trivial amount of code. Concerns such as loading samples, basic error handling, caching of inference results, or creating AST for common TypeScript construct could be reused between providers for different formats.

For this reason, we should implement a support library containing utility code and a reasonable factory/builder API for creating additional type providers.

R19 A library will be implemented to facilitate the development of type provider packages with a common sample-based workflow.

Apart from implementing the general type provider mechanism, several type provider packages will be developed to showcase that the overall solution works and to evaluate the requirements we specified in this chapter.

We selected CSV, XML, and JSON as the initial formats to work with for their widespread use and different data layouts. Because JSON is a "native" format in JavaScript (almost 1:1 mappable to a JS object), the JSON provider should focus on generating useful run-time validation code instead of trivial parsing. The run-time data-access code generated by the implemented providers will be compatible with the Node.js runtime.

R20 Showcase type provider packages for CSV, XML, and JSON will be implemented.

3. Solution documentation

This chapter presents the prototype implementation of the type provider mechanism that was realized during the thesis.

First, we provide some background for the TypeScript compiler and describe how we integrated the type provider feature. After that, we deal with the implementation of type provider packages supported by our compiler.

The code artifacts for the thesis can be found in the attached ZIP file. See Attachment A.1 for an overview of the submitted files.

3.1 Compiler integration

The TypeScript compiler is an open-source project maintained by Microsoft [26]. It is itself written in TypeScript and is typically used in two modes. First, as a batch command-line Node.js application (invoked by the `tsc` command). Second, as a long-running service with a JSON API (the `tsserver` program) used by a code editor to provide IDE features to the user.

In the abstract, the architecture of the core TypeScript compiler can be described as a traditional linear pipeline illustrated in Figure 3.1:

- The *program* gathers the files to be compiled by starting from a set of root files and transitively following their static imports.
- The *scanner* splits the input source text into tokens.
- The *parser* takes the tokens and builds up the abstract syntax tree while reporting syntax errors.
- The *binder* takes the AST and creates tables of Symbols that bind together different locations related to the same declaration.
- The *checker* takes the Symbol tables and performs static type checking on the AST.
- The *transformers* rewrite the AST to remove or replace syntax unsupported by the target version of JavaScript.
- The *emitter* writes the AST into a source text.

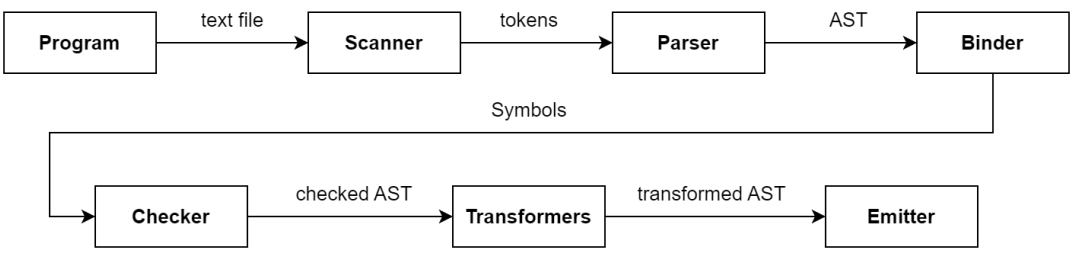

Figure 3.1 An abstracted view of the compiler pipeline

The actual implementation of the current compiler is organized quite differently and follows a "pull-based" rather than "push-based" model [27, 28]. This means, for example, that the emitter drives the checker, which drives the binder. All operations are performed as on-demand, potentially partial tree walks over the mostly stateless AST. This design is motivated by the goal of supporting the language service and advanced IDE features rather than the high throughput of the batch command line compiler.

3.1.1 Inserting the type provider mechanism

The key question for implementing type provider support is how do type providers fit into the existing compiler architecture described in the previous section. Can we somehow *insert* the mechanism into the current pipeline without a significant rewrite of the compiler? And if so, in what part of the pipeline?

The most promising approach seems to be to treat type providers as a virtual source of parsed AST and feed that AST into the later phases of the compiler. This is based on the following reasoning:

1. Ultimately, our goal is to both type check code that uses provided types and emit the actual provided code so that it can be used during run-time.

2. The checker uses the AST heavily – both directly and indirectly via the Symbol tables produced by the binder, which are, in turn, built from the AST. If we wanted to provide the internal Type representations instead of AST, we would likely need to introduce a significant amount of alternative code paths in the checker that do not depend on existing AST.[1].

3. The emitter and the transformer pipeline work exclusively with AST. Therefore, we would still have to synthesize the AST at some point.

4. Implementing type providers as generators of AST also means (as discussed in Section 2.5) that type providers can work with the already public parts of the compiler API and do not require exposing additional internal functionality.

Because the checker and the emitter are designed around `SourceFile` nodes as the basic *input* unit of work, it makes sense to design type providers with the `SourceFile` as the *output* unit of work. This is also supported by our proposed provided import syntax (presented in Section 2.3).

Currently, `SourceFile` nodes get introduced in the pipeline either as a result of parsing a root file or when processing imports of the current file. Therefore, we can generally treat provided imports as regular imports until the point where the regular workflow tries to retrieve the `SourceFile` node of an imported file. Normally, a source file is loaded from the disk and parsed into AST, or a previously parsed `SourceFile` node is retrieved from a cache. We can use the information that the import is provided (which we know unambiguously from the use of the `provided` modifier) and instead return a virtual `SourceFile` created by the type provider.

[1]For context, the checker is by far the most complex component of the compiler with over 50 thousand lines of code currently [29]

3.1.2 Adding provided imports

As discussed in Section 2.3, we introduced a new syntax to TypeScript for specifying provided imports. The provided import declaration looks like this:

```
1 import provided * as Xml from "xml-provider" with { sample: "..."
    };
```

Adding support for this new construct has been straightforward. We modified the `parseImportDeclaration` function in the parser to add a look-ahead check for the presence of the `provided` modifier keyword. If the keyword is found, the `isProvided` flag is set on the `ImportDeclaration` node. We decided to add the flag instead of defining a new kind of node because, in most contexts, the provided import should be handled the same way as a regular import declaration.

We also modified the `ImportAttributes` interface and type checking for the respective AST node to allow for non-string values in the import attributes object literal (the `with { ... }` block). This allows for passing string, numeric, and boolean option values.

The `isProvided` flag on the `ImportDeclaration` node is used later when collecting a list of imported modules for the source file containing the import declaration. We treat the import mostly the same as an import from the actual type provider package. However, when the compiler tries to retrieve the declaration file for the provider package, instead of loading the physical file from disk and creating AST from that with the `parseSourceFile` function, we defer to the type provider mechanism by calling the `createProvidedSourceFile` function.

When handling the provided `SourceFile` and the virtual module it represents, we uniquely identify it with a special *import hash*. We compute this hash based on the name of the provider package, the content of the import attributes of the provided import statement, and the directory path containing the importing source file. We use these values to ensure that multiple provided imports with different configurations (or with different relative path bases) get treated as different modules while provided imports with the same configuration get reused.

3.1.3 Type provider invocation

When a provided source file is requested through the `createProvidedSourceFile` function, the following actions take place in our `invoker` module:

1. Name of the provider package and path to its entry point file are resolved.
2. The type provider module is dynamically loaded using the `require` function of the Node.js module loader. (The modules are loaded only once and then cached.)
3. The invoker checks if the module's exports contain at least one of the functions `provideSourceFileSync` or `provideSourceFileAsync` (in this order of priority).
4. The provider function is called in a `try/catch` block. The invoker passes in import options read from the `ImportAttributes` block of the `ImportDeclaration` block and information about the compilation context.

Code Listing 3.1 The type provider interfaces

```
export interface SyncTypeProvider<TOptions extends object> {
  provideSourceFileSync(options: TOptions, context:
      ProviderContext): TypeProviderResult
}

export interface AsyncTypeProvider<TOptions extends object> {
  provideSourceFileAsync(options: TOptions, context:
      ProviderContext): Promise<TypeProviderResult>
}

export interface ProviderContext {
  importingFilePath: string;
  importHash: string;
  runtimeTarget?: "browser" | "bun" | "deno" | "node";
}

export interface TypeProviderResult {
  sourceFile?: SourceFile;
  generalDiagnostics?: DiagnosticMessage[];
  optionDiagnostics?: Map<string, DiagnosticMessage[]>;
  requiresAsync?: boolean;
}
```

5. The provider can return a `SourceFile` AST node with the provided code and a list of diagnostics to be passed to the user to indicate issues encountered by the provider (e.g., unavailable sample).

6. The provided source file (if any) is returned to the compiler.

We omitted error checking, which is performed between each step. If the provider module cannot be loaded, does not satisfy the expected interface, or throws an error during execution, the invoker creates an error diagnostic displayed on the provided import declaration. The same is done when the provider call succeeds but returns diagnostics produced by the type provider.

The TypeScript interfaces specifying the API between the compiler and the type providers are shown in listing 3.1.

3.1.4 Transformers and emitting JavaScript

Unlike other compilers, the TypeScript compiler does not produce machine code or byte code but a JavaScript source text. Because TypeScript is mainly designed as a syntactic superset of modern JavaScript, this process is done in a series of AST rewrites performed by modules called transformers, followed by simple printing of the final AST.

A new transformer was added to the existing pipeline to handle the provided import statements. This transformer gets executed second in the pipeline, after

the basic transformer responsible for erasing TypeScript-only features. We chose this order because the TypeScript transformer also handles erasing unused and type-only imports. This way, we don't have to differentiate the two or remove potentially empty imports.

The type provider transformer does three things:

1. Removes the `provided` modifier keyword.
2. Removes the import attribute block containing provided import options (`ImportAttribute` node in the AST).
3. Rewrites the module specifier of the import (the reference in double-quotes) so that instead of the name of the type provider package (e.g., `"@ts-providers/csv"`), it contains a relative path to the correct emitted file containing the imported provided code.

The path written into the import specifier in the third step is resolved based on the relative path of the transformed TypeScript source file in the compilation's `rootDir`. This is done because we emit the provided source files into a subdirectory named `"_provided"` under the compilation's `outDir`.

Each emitted provided source file uses its import hash (see 3.1.2) as the file name. We also add one of the extensions `.mjs` or `.cjs` depending on the target module system (ESM or CommonJS, respectively).

3.1.5 Open issues

This section briefly discusses two issues that need to be resolved in a production-ready, non-prototype implementation of type provider support.

Source file management

Our solution integrates the type provider mechanism into the compiler pipeline by resolving module imports based on provided import statements using virtual `SourceFile` nodes instead of parsing physical source files stored on the disk. However, in several subsystems of the compiler, it is assumed that each source file is uniquely identified by a string representation of its path in the host file system, for example, for caching).

We want to properly support multiple simultaneous provided imports or a combination of provided and non-provided imports from the same type provider package. Therefore, we need to do one of the following:

1. Modify or substitute all relevant subsystems to not rely on file system paths file identifiers in case of provided source files.
2. Give unique names to the provided source files to be as compatible with the existing code as possible.

We spent a significant amount of time trying to implement the first variant. However, it has proven to be too large of a change for the limited scope of our

thesis. Consequently, we implemented a unique naming pattern for the provided files. We modified the existing code only so that it does not try accessing the non-existing files in the file system.

The naming pattern used in the current prototype is as follows:

```
Provided|{file path of the provider package entrypoint}|{import hash}
```

This solution works rather well during batch compilation with the command-line compiler. Each provided import is resolved only once. When the user's code contains multiple provided imports with the same options and context, the provided `SourceFile` gets reused.

However, when running in the language server mode, there are remaining issues with source file management that forced us to disable the language service capabilities for incremental parsing. This leads to more full regenerations of the provided `SourceFile` and, therefore, more invocations of the type provider than needed. With inference caching implemented in the type provider, the developer experience is not hindered, at least for the size of the codebases that we tested. However, this is still an apparent inefficiency that should be avoided and probably limits the solution's scalability to large code bases and data samples.

Asynchronous provider invocation

Certain operations, particularly network operations (HTTP requests, database queries, etc.), do not have synchronous (blocking) implementations in JavaScript runtimes such as Node.js. The core TypeScript compiler is designed and implemented as an entirely single-threaded, synchronous application. This means it is impossible to support asynchronous type providers (that return a `Promise` instead of an immediate result) without some workaround.

Due to its fundamentally single-threaded runtime model, JavaScript supports asynchronous execution only via sequentially processed event loop [30]. This means that a caller cannot simply wait for the result of an asynchronous operation, as that would block the processing of the event loop. This is true regardless if the asynchronous operation is done via traditional callbacks, with the ES6 Promise API, or with `async`/`await`.

The only functional solution with an acceptable performance overhead that we found is using the `deasync` package (or some variant of the same approach). The `deasync` package is loaded as a native Node.js plugin. It exposes the ability to interact with the native event loop processing in Node.js to the interpreted JavaScript code. Using that, it is possible to block the caller from continuing without stopping the entire event loop.

However, `deasync` and similar Node.js plugins have been described as unsafe and unsupported [31]. Depending on such a method (that can be broken in future Node.js versions) in a production release is not feasible for a project such as the TypeScript compiler.

We experimented with invoking the asynchronous type provider in a Node.js worker thread [32] and waiting for the result using `Atmomics.wait` [33]. However, due to the need to serialize and deserialize arguments and results (and possibly inherent inefficiency in the worker thread mechanism), this solution has been

shown to have too much performance overhead to be viable. However, further investigation should be done in this regard.

3.2 Provider implementation

In the previous section, we discussed infrastructural support for type providers in the compiler. We also specified the interface that the compiler expects each type provider to implement (see Section 3.1.3). In this section, we move to the implementation concerns of the providers themselves.

We first discuss several common tasks that a typical provider needs to handle, such as retrieving samples, code generation, and caching. We present the Core library created to support the implementation of such providers.

In the rest of the section, we describe three provider packages that we implemented to showcase and evaluate the solution's viability. We discuss the inference algorithm and code generation specifics of each provider. We chose CSV, XML, and JSON as the showcase formats for their wide use and reasonably different structure.

All of the example providers were implemented using the Core provider builder and share the same infrastructure described in Section 3.2.1.

3.2.1 Core library

The Core library is distributed as a standalone package `@ts-providers/core`. It contains a collection of utility types and functions that can be used to streamline the implementation of type provider packages. Some of these can be used separately, including utilities for naming conversions, primitive value parsing, and gathering diagnostics.

Provider builder

The main feature of the library is the provider builder. The builder's API consists of a generic interface `BasicProviderFunctions` and a factory function `createBasicBuilder` that, given a small set of functions, constructs the complete type provider implementation.

We used the provider builder to implement all three of our showcase type providers. Code listing 3.2 illustrates the use of the builder to implement the JSON-Zod provider. A step-by-step tutorial for using the builder to create a new type provider is given in Section 4.3.

Code Listing 3.2 Using the provider builder

```
1  const providerFunctions:
       BasicTypeProviderFunctions<BasicTypeProviderOptions,
       JsonTypeDescriptor> = {
2      infer: (textInput: string): JsonTypeDescriptor => {
3          const parsed = JSON.parse(textInput);
4          return inferJsonTypeDescriptor(parsed);
5      },
6      generateCode: (rootType: JsonTypeDescriptor) => [
7          ...generateImports(),
8          generateSchema(rootType),
9          generateTypeExport(),
10     ],
```

```
11 }
12
13 export const JsonZodTypeProvider =
       createBasicTypeProvider(providerFunctions);
```

The provider implementation created by the builder takes care of several shared concerns.

Loading samples

The provider implementation supports inline samples, file-based samples, and HTTP-based samples. Files are read using the `fileReadSync` function from the `node:fs` namespace. HTTP samples are retrieved using the `fetch` API supported since Node 18. Note that the builder generates both a synchronous and asynchronous variant of the provider function. However, only the HTTP sample is loaded asynchronously to avoid the overhead of the compiler actively waiting for the result of an asynchronous operation.

Error handling

The provider implementation emits a set of standard diagnostic messages covering situations such as missing sample configuration or errors during sample retrieval. The individual provider can add its own diagnostics (see below).

Caching

The provider's inference results are cached and reused in further provider invocations. The import hash (see Section 3.1.3) is used as the basis for the cache key. This ensures proper invalidation for provided imports with inline samples (as those are included in the hash). For file-based samples, the provider implementation adds the last modification timestamp of the file so that the inference is re-run when the sample file changes.

The builder interface

The `createBasicProvider` function expects an instance of the `BasicTypeProviderFunctions` interface. This interface has two generic type parameters:

1. Model type for the options supported by the type provider. The built-in `BasicTypeProviderOptions` interface can be used if no additional options are needed. It contains standard options for specifying the sample as a string, file path, or HTTP URL.
2. Type of the object produced by the `infer` function and consumed by the `generateCode`.

In order to satisfy the interface, these functions need to be implemented:

- `infer` — This function receives the sample content as a string and returns an instance of the generic `InferenceResult` type. When successful, this result is cached in the provider-local cache.

- `generateCode` — This function receives the inference result that has been either newly produced by the `infer` function or retrieved from the cache. Based on the result, the function generates the provided code as an array of statement AST nodes.
- `parseCustomOptions` — When a provider uses custom import options, this function needs to be implemented to parse these options and set default values for options not set in the provided import statement.

Each function is also given an instance of import options, the provider context record (see Section 3.1.3), and an instance of `DiagnosticCollector` that can be used to emit diagnostics that will be returned to the compiler once the provider finishes or one of the functions indicates irrecoverable failure by returning an `undefined` value instead of its result.

3.2.2 Code generation

Regardless if a type provider is implemented using the builder API or from scratch, a key functionality it needs to handle is generating TypeScript code in the form of abstract syntax tree nodes defined by the public compiler API [34]. The compiler API exposes both the interfaces for various AST nodes and factory functions for creating the nodes. Code listing 3.3 shows an example of using the factory methods to construct AST for a simple TypeScript statement.

Code Listing 3.3 Constructing AST with compiler API

```
import { factory, NodeFlags } from "typescript";

// This creates the statement: const answer = 42;
const node = factory.createVariableStatement(
  undefined,
  factory.createVariableDeclarationList(
    [factory.createVariableDeclaration(
      factory.createIdentifier("answer"),
      undefined,
      undefined,
      factory.createNumericLiteral("42")
    )],
    NodeFlags.Const
  )
);
```

The AST nodes are designed to be generally immutable, and most of their fields are declared as `readonly`. The factory methods therefore need to be able to set all of the node's fields to account for all possible configurations. Due to this, they are often needlessly verbose for common scenarios. From our experience, this verbosity quickly adds up as the complexity of the code to be generated grows.

For this reason, we implemented a set of helper functions for the most common operations, such as creating primitive type literals, variable or property assignments, type nodes, and so on. These functions are exported by the `Core` library in the `factoryHelper` object.

Developers of type providers can also utilize the `ts-morph` library [22], which implements a convenient builder API for creating TypeScript AST. It also exposes the compiler's parsing capabilities and lets you create AST from a string. (This feature is also possible with the official compiler API, albeit in a rather contrived way.)

In some situations, constructing AST from a string can be a significantly more user-friendly approach than using the factory methods. The developer can simply write a template for the resulting code and fill it with dynamic values such as identifiers and value literals. However, we performed a rudimentary performance comparison between the factory method-based and string-based approaches in which the latter were shown to be about 20 to 50 times slower, or single digits of milliseconds vs tens of milliseconds in absolute numbers for our benchmark AST. More rigorous evaluation should be done. However, the difference was significant enough for us to opt for factory methods when implementing our showcase type providers.

Generating interfaces

Generating interfaces or type alias declarations is a common requirement for type providers. Depending on the complexity of the types, particularly the level of nesting, it can become quite complex when using the compiler API. For this reason, the `Core` library exports another abstraction in the form of `TypeDescriptor` and `DeclarationDescriptor` classes. These types can be used to build a simplified intermediate tree representation of types. The tree is then able to create its matching TypeScript AST signature recursively.

3.2.3 CSV provider

The first implemented type provider is for the CSV data format. The provider is distributed in the `@ts-providers/csv` package. Its use is documented in Section 4.2.3.

CSV (Comma-separated values) is a simple plain text format for tabular data [35]. It is commonly used for transferring data between systems and databases or in data science. When working with CSV data in a TypeScript program, it is natural to parse it as a sequence or array of row records.

The CSV type provider tries to facilitate access to the CSV data that is both type-safe and practical. It infers the most specific type for each column based on the values seen in the sample. From this, the provider generates a TypeScript interface describing an individual row record and a set of data access functions that can be used to load and parse CSV files with compatible data. The data is validated when read to ensure that the run-time types match the inferred static types.

Figure 3.2 shows a provided CSV type supporting the autocomplete feature in VS Code.

Inferring row type

When trying to infer types for CSV data, a few facts need to be considered:

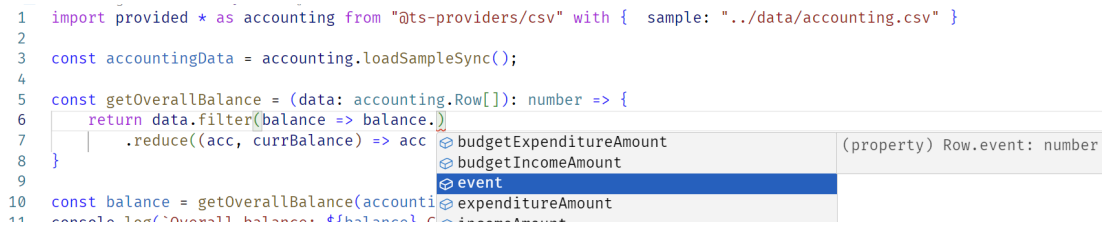

Figure 3.2 Autocompletion of members from inferred CSV type

- The CSV format itself does not carry explicit typing information.
- All values are implicitly strings. There is no standardized syntactic difference between e.g. string, numeric, or boolean values.
- The columns can be optionally labeled using the first row as a header.
- Rows can have missing values in any of the columns.
- There are competing conventions for representing special values. For example, booleans are sometimes represented by a pair of special values such as `"true"/"false"`, `"Y"/"N"`, `"1"/"0"`, and so on.

We decided to implement a pragmatic inference algorithm similar to [6]. The algorithm determines the most specific type for each column that can be used to safely represent all values seen in the sample data for that column.

CSV kind	TS type	Allowed values
Bit	boolean	One of “1” or “0”
Bool	boolean	One of “true”, “false”, “y”, “n”, “yes”, “no”
Int	number	Returns true for `Number.isSafeInteger`
Float	number	Does not return `NaN` for `Number` constructor
BigInt	bigint	Does not throw for `BigInt` constructor
Date	Date	Valid ISO 8601 date string
String	string	Any string

Table 3.1 Mapping of CSV values to TypeScript types

Table 3.1 shows the supported CSV value kinds and their mapping to TypeScript types. During inference, we consider each column to have one of the kinds based on the values seen so far. We go through all of the sample rows, and we check whether the column’s current kind is compatible with the value in the current row. If not, the column kind can change according to the relation rules shown in Figure 3.3.

Note that the `String` kind represents the top type of the hierarchy. That is, any column can be inferred as a `String`, and once a column is inferred as such, the result cannot change.

A column is considered nullable if some sample row contains a null value, where by convention, we consider as null not only an empty string but also a set of special strings (e.g. `"null"`, `"n/a"`).

It should be noted that the current parsing code for the CSV provider does not account for situations such as separator characters inside quoted values. For

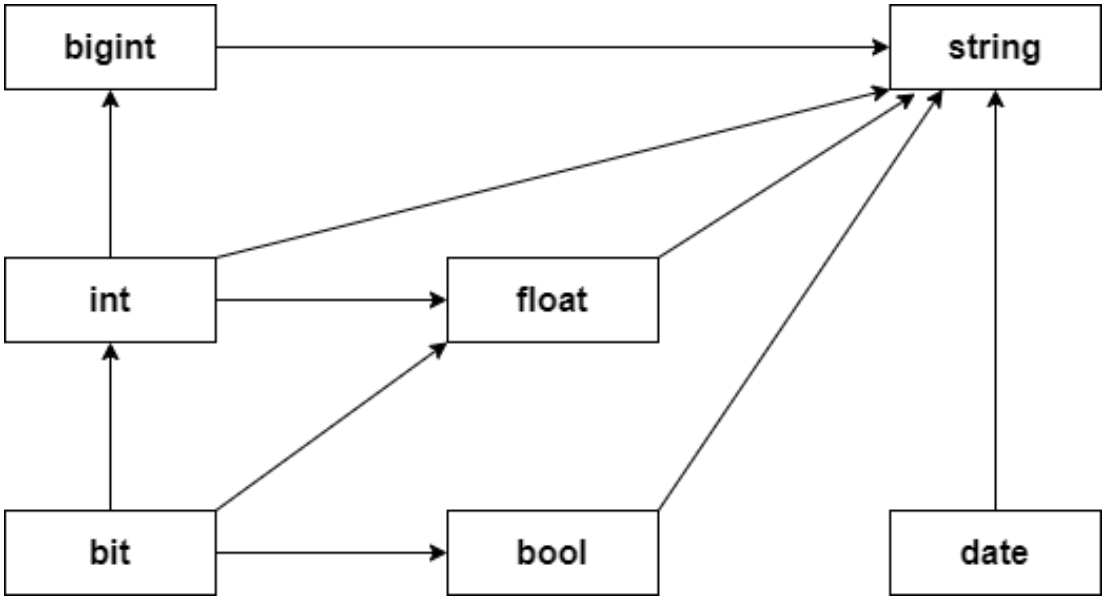

Figure 3.3 Relation between the value kinds for CSV type inference algorithm

production use, we would replace our demo implementation with a more robust one or use an existing library.

Code generation

After successful inference, the CSV provider generates TypeScript AST for the following components:

- A `Row` interface declaration. For each column from the sample, there is a matching field with the type inferred for that column. Names of fields are converted to camel case (unless disabled by import option). If the sample did not have a header row, dummy names `"column$n"` are used instead.
- A `parseRow` function declaration. The function handles parsing string values from CSV input according to the inferred column types and throws an error for incompatible values.
- A `header` array declaration containing name mapping for known columns from the sample. This allows reading data files with different order of columns or with additional columns.
- A `provider` object declaration. The `provider` is an object that contains all of the generated data access functions. It is created during run-time when the provided module is loaded using a factory function. This significantly decreases the amount of code that needs to be generated code as the factory functions are contained as actual code in the provider package itself.
- An import statement that imports the external functions and types used in the generated code.

Code listing 3.4 contains an example of code generated by the CSV provider from a file-based sample.

Code Listing 3.4 Example of code generated by the CSV provider

```
import { createProvider, valueParser, type CsvProvider, type
    CsvColumnMapping } from "@ts-providers/csv";
export interface Row {
  item: string;
  amount: number;
  isBlue: boolean;
}
export const header = [
    { name: "item", originalName: "Item" },
    { name: "amount", originalName: "Amount" },
    { name: "isBlue", originalName: "Is blue" }] as const;
export const provider: CsvProvider<Row> = createProvider(parseRow,
    ",", "utf8", header, "../data/01.csv", undefined, undefined);
function parseRow(row: string[], rowIndex: number, columns:
    CsvColumnMapping[]): Row {
  return {
    item: valueParser.parseString(row[columns[0].position],
        columns[0].position, rowIndex, columns),
    amount: valueParser.parseCsvValue(valueParser.parseInt, row,
        columns[1].position, rowIndex, columns),
    isBlue: valueParser.parseCsvValue(valueParser.parseBool, row,
        columns[2].position, rowIndex, columns)
  };
}
```

3.2.4 XML provider

The second implemented type provider is for the XML data format [36]. The provider is distributed in the `@ts-providers/xml` package. Its use is documented in Section 4.2.3.

XML (eXtensible Markup Language) is a plain text data format that represents data in a hierarchical structure of elements. It is widely used for serialization, data transfer, and configuration files. An XML document is a rooted tree of tag-based elements which can contain attributes and nested child elements.

The XML type provider aims to provide a more JavaScript-native interface to access XML data. It does that by parsing the XML tree into a record with properties representing the element's attributes and child elements. From this, the provider generates a set of TypeScript interfaces describing the elements in the sample, and a set of data access functions that can be used to load and parse compatible XML documents. The data is validated when read to ensure that the run-time types match the inferred static types.

It should be noted that the XML provider infers the type from samples of actual XML documents and does not rely on any schema for XML, such as XML Schema [37].

Inferring XML document shape

The XML type provider tries to represent the tree of XML elements as a tree of TypeScript objects. However, there are innate differences between the representations that need to be solved.

Firstly, an XML element has two separate groups of "properties", attributes, and nested child elements. One option would be to merge these together as fields of the TypeScript type. However, this could lead to name conflicts (requiring name mangling) and would mean that the user loses potentially valuable information, whether some property comes from an attribute or a sub-element. For this reason, we store the attributes into a special `attributes` field (with backup name mangling). Child elements are stored as regular top-level fields.

Secondly, an XML element can have multiple child elements with the same name. This means we cannot simply use the child element name as a key in the parent record. We deal with this by checking whether, in some occurrence of the parent element in the sample document, it contained multiple instances of said child element. If not, we represent the child in the parent element's TypeScript type as a simple field with the child's recursively inferred type. If yes, we represent the child as an array field.

The solution with arrays creates an interesting problem for run-time validation. If an element E contained only a single instance of child C in the sample data, we infer it as a simple, single-value field in the parent. What should happen in the run-time if we read a different document that has an element E with multiple child instances of C?

One way to deal with this would be to violate the static type guarantees and load the children into an array. A different solution would be to honor the static typing and only read the first encountered instance of C and discard the rest. Both of these solutions have downsides that we consider too significant. Therefore, in such a situation, we load the first instance of C into the appropriate field `c`, and we load the other instances of C into a special field with the mangled name `#c`. This field is guaranteed not to collide with other fields because XML names cannot contain the character `#`, but it can be accessed in JavaScript using the brackets operator. Although this is not an optimal user experience, it does not violate type safety, and the user can still check and access the full data.

Similarly to the CSV provider (see Section 3.2.3), we also infer the most specific type for values of attributes and simple (leaf) elements. Table 3.2 shows the supported CSV value kinds and their mapping to TypeScript types. The relationships between the value kinds are illustrated by Figure 3.4. An attribute or element is considered optional if it does not occur in all instances of an element E in the sample. All attributes and children of the root element are optional because there would be no way how to represent their optionality in the sample.

The XML provider uses a third-party library `xmldoc` [38] to handle the parsing of the raw text input into a tree structure of string-typed nodes. However, the inference process described here is performed fully by our code.

Code generation

After successful inference, the XML provider generates TypeScript AST for the following components:

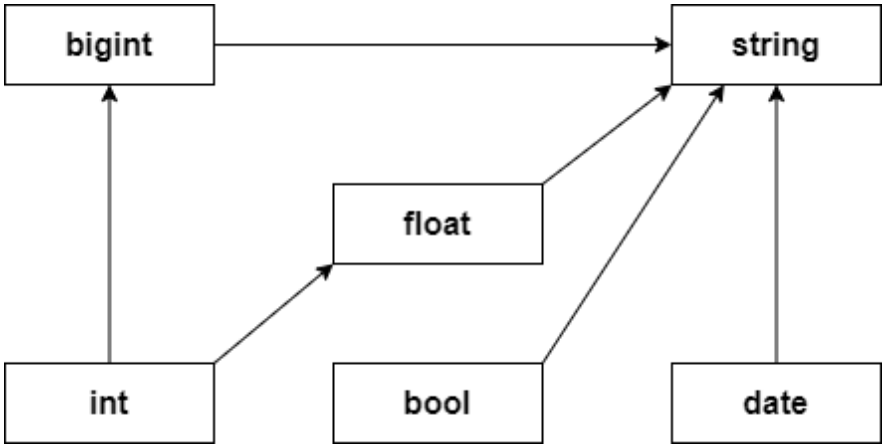

Figure 3.4 Relation between the value kinds for XML type inference algorithm

XML kind	TS type	Allowed values
Boolean	boolean	One of “true” or “false”
Int	number	Returns true for `Number.isSafeInteger`
Float	number	Does not return `NaN` for `Number` constructor
BigInt	bigint	Does not throw for `BigInt` constructor
Date	Date	Valid ISO 8601 date string
String	string	Any string

Table 3.2 Mapping of XML values to TypeScript types

- A set of interface declarations named after elements found in the sample. For each attribute and child element of an element, there is a matching field with the type inferred for that attribute or child. Names of types and fields are converted to Pascal case and camel case, respectively (unless disabled by import option). If an element only appears as a leaf in the sample, it is inlined in its parent elements and does not get its own interface.
- A `typeMap` object declaration. This object stores a subset of information from the inference process that can be used to validate the structure and types of an XML document during run-time parsing.
- A `provider` object declaration. The `provider` is an object that contains all of the generated data access functions. It is created during run-time when the provided module is loaded using a factory function. This significantly decreases the amount of code that needs to be generated code as the factory functions are contained as actual code in the provider package itself.
- An import statement that imports the external functions and types used in the generated code.

Code listing 3.5 shows an example of code generated by the XML provider from an inline sample.

Code Listing 3.5 Example of code generated by the XML provider

```
1 import { createProvider, type XmlProvider } from
    "@ts-providers/xml";
```

```
export interface Library {
    book?: Book[];
}
export interface Book {
    attributes: {
        read?: boolean;
    };
    title: string;
    pages?: number;
}
export const typeMap = {
    library: {
        children: { book: { isArray: true } }
    },
    book: {
        attributes: { read: { valueKind: 1 } },
        children: { title: { isRequired: true }, pages: {
           valueKind: 2 } }
    },
    title: {},
    pages: {
        valueKind: 2
    }
};
const inlineSample = "<library><book
   read='true'><title>Odyssey</title><pages>300</pages></book><book><title>Illi
export const provider: XmlProvider<Library> =
   createProvider(typeMap, false, false, "utf8", "", undefined,
   inlineSample);
```

3.2.5 JSON-Zod provider

The third implemented type provider is for the JSON data format [39]. It is distributed in the `@ts-providers/json-zod` package. Its use is documented in Section 4.2.3.

The JSON (JavaScript Object Notation) format has a special position in the JavaScript and TypeScript ecosystem. Since JSON's implicit type system is a subset of the JavaScript type system, parsing JSON into JavaScript objects is straightforward and is covered by the standard `JSON.parse` function. Modern implementations of JavaScript also support statically importing JSON documents as object literals using the ESM import syntax. TypeScript is then able to infer the static type of such document as shown in code listing 3.6.

Code Listing 3.6 Native static import of JSON

```
    import MyData from "./my-data.json" with { type: "json" };

    type MyDataType = typeof MyData;
```

This makes implementing a JSON type provider in the same style as we did with CSV and XML obviously less useful. However, these features only cover static imports of JSON files and do not cover dynamic loading or provide any run-time validation. Because JSON is commonly used for transferring data between clients and services, run-time validation is a very important issue. The standard solution is to use one of the mature run-time validation libraries such as ajv [40] or Zod [41].

The Zod library is notable for its ability to infer the static type of the validated object from the schema object (see line 16 in code listing 3.7) using just the built-in type inference capabilities of TypeScript. This avoids the redundancy of writing both the target type and the validation schema. Code listing 3.7 shows how one can use the Zod library [41] to define a schema for a JSON document and validate an object with the schema.

Code Listing 3.7 Validating JSON data with a Zod schema

```
1  import { z } from "zod";
2  const librarySchema = z.object({
3    libraryName: z.string(),
4    location: z.object({ city: z.string(), country: z.string() }),
5    books: z.array(
6      z.object({
7        title: z.string(),
8        author: z.string(),
9        year: z.number(),
10       inStock: z.boolean()
11     })
12   ),
13   tags: z.array(z.string())
14 });
15
16 type Library = z.infer<typeof librarySchema>;
17
18 const value: Library = librarySchema.parse(...);
```

The Zod library is notable for its ability to infer the static type of the validated object from the schema object (see line 16 in code listing 3.7) using just the built-in type inference capabilities of TypeScript. This avoids the redundancy of writing both the target type and the validation schema.

However, with type providers, we can do better than that and generate the schema itself from a representative sample. In this way, type providers can utilize the potential of existing libraries while increasing their productivity.

To demonstrate such an approach, we implemented the JSON-Zod type provider (available in the `@ts-providers/json-zod` package). This provider reads a sample of JSON data and generates a Zod schema that can be used to both statically type and run-time validate JSON data. Code listing 3.8 shows usage of the JSON-Zod provider.

Code Listing 3.8 Using the provided JSON schema

```
import provided { schema, SchemaType as Library } from
   "@ts-providers/json-zod" with { sample: "./libraries.json" }

const value: Library = librarySchema.parse(...);
```

Inferring JSON document shape

Thanks to the inherent compatibility of type representations in JSON and TypeScript, the inference solution in the JSON-Zod provider is notably simpler than in the previous two cases. The provider uses the standard `JSON.parse` function to read the specified sample. Then it recursively walks through the parsed object, checks the run-time type of each element with the `typeof` operator, and builds up an intermediate tree representation of types. For `object` fields, it separately handles nested objects, arrays, and `null` literals. We also check strings as possible (ISO 8601 formatted) Date values.

Later a second tree walk is performed on the type tree, which is used to generate the TypeScript AST, which represents the Zod schema declaration. The schema and the inferred type of the validated object are then provided to the user. Unlike the other two providers we intentionally do not generate data access functions in the JSON-Zod provider. Parsing and validation can be performed using a combination of the standard `JSON.parse` function and the provided schema. Retrieving the data itself can be done by the user's preferred means for the runtime environment they are targeting. As a consequence, unlike our other two providers, the JSON-Zod provider is fully compatible with all modern JavaScript runtimes and browsers.

4. User documentation

This chapter contains an overview of how to use individual parts of the solution. First, we give instructions for setting up the development environment to use the type provider-enabled TypeScript compiler on the command line and in Visual Studio Code. Then we describe the use of the implemented type provider packages. Lastly, we show how to implement a new type provider package in the form of a step-by-step tutorial.

4.1 Using the modified compiler and language service

To run the modified TypeScript compiler, first install the Node.js runtime following the instructions at `https://nodejs.org/en/download/`. Our fork of TypeScript 5.5 has been tested on Node 20 LTS, however, any version since Node 18 should suffice. The installation also comes with the NPM package manager CLI.

Any operating system platform supported by the Node runtime should be compatible with the compiler. However, testing was performed only on Windows 10 x64.

4.1.1 Getting the compiler

In this section, we describe three methods of installing and using the modified TypeScript compiler.

Local NPM installation (recommended)
To use the compiler in a specific TypeScript/JavaScript project, you can install it as a normal NPM package:

```
1 npm install --save-dev @ts-providers/compiler
```

This adds the modified TypeScript distribution to the `devDependencies` section of the project's `package.json` file (or creates such file first if in an empty directory).

You can now use the `tsc-providers` command in the context of the project. This is a renamed variant of the standard `tsc` command from the mainline TypeScript distribution and has the same set of options (see documentation [42]). In the project directory, you can either:

- Invoke the compiler directly using `npx tsc-providers`,
- Use `tsc-providers` in commands defined in the `scripts` section of `package.json`. For example, you can set up a 'build' script for your project and call it with `npm run build` as shown in listing 4.1.

To get more information about setting up JavaScript and TypeScript projects in general, including the `package.json` and `tsconfig.json` files, please consult the documentation [43, 44]. You can use the configuration in listing 4.2 as a start.

Code Listing 4.1 Example of a minimal package.json configuration

```
{
  "name": "my-project",
  "devDependencies": {
    "@ts-providers/compiler": "^5.5.3-providers-v13",
  },
  "scripts": {
    "build": "tsc-providers -p tsconfig.json"
  }
}
```

Code Listing 4.2 Example of a minimal tsconfig.json configuration

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "declaration": true,
    "outDir": "./dist" // Always set outDir with type providers
    },
    "include": [ "src/**/*.ts" ]
}
```

The only key setting in `tsconfig.json` is `outDir` which **has to be set in order for the type providers to emit run-time code**. If you only care about static type checking, you do not need it.

Note that you should use NPM to install the compiler package, even if you are using a different compatible runtime such as Bun.js [45]. This is to ensure that the install scripts of the compiler's dependencies are executed. Bun's package manager, in particular, is known to disable these unless overridden with a special configuration.

Global NPM installation
You can also make the command `tsc-providers` available globally by executing:

```
npm install -g @ts-providers/compiler
```

Because the compiler binary has been renamed to `tsc-providers`, it can be installed alongside the standard TypeScript compiler, which is invoked using the `tsc` command.

Build from source
To build the compiler locally from the source, simply enter the `compiler` directory in the Attachment and execute:

```
npm install
hereby local
```

The build output can be found under the `built/local` subdirectory. Executing `node ./build/local/tsc.js` runs the same program as the one distributed in the NPM package.

To compile a single file, you can then invoke the compiler like this:

```
node ./build/local/tsc.js ../path/to/some/source/file.ts
```

To compile an entire TypeScript project, run:

```
node ./build/local/tsc.js -p ../path/to/some/project/tsconfig.json
```

4.1.2 Setting up Visual Studio Code

As mentioned previously, the only supported IDE currently is the Visual Studio Code [46]. Other editors compatible with the TypeScript Language Server might work but have not been tested.

Visual Studio Code has built-in support for TypeScript, which comes with its own TypeScript installation. This means that when you are editing TypeScript source files, the editor is by default using the language service from a different installation than the one you might be using to build your project (typically installed as a local development dependency, see *Local NPM installation* above).

In order for VS Code to use the locally or globally installed modified compiler, you have to do two things.

First, create or modify the file `.vscode/settings.json` under the root of your workspace and add the following options:

```
"typescript.tsdk": "./node_modules/@ts-providers/compiler/lib",
"typescript.enablePromptUseWorkspaceTsdk": true
```

Second, restart the editor and open any TypeScript file in the project. Accept the pop-up prompt that should appear in the bottom-right corner asking you if you want to switch the editor to use the workspace version of TypeScript. Alternatively (or if the pop-up does not appear), open the command menu (by default bound to the key F1) and find and press the command `TypeScript: Select TypeScript version...`, then select the `Use Workspace Version` option. In both cases, the version indicated should be labeled `5.5.3-providers-v...`.

If you have issues with the compiler not being available, please run `npm install` first and restart VS Code.

Note on incompatible VS Code extensions
During later testing, we found that the *Vue - Official* extension is incompatible with our modified language service and causes it to malfunction (even if the project does not use Vue at all). We have not been able to determine the cause of this conflict. We can't also rule out that there are other extensions with similar issues.

If you can compile source files with provided import on the command line but see errors with provided imports in VS Code (in particular with the message "Module could not be loaded as type provider..."), please check whether you have the Vue extension (or similar TypeScript related extension) installed, and please check if uninstalling the extension resolves the issue.

4.2 Using type provider packages

Once the development environment is set up, you can start to use existing type provider packages. There are some requirements for the configuration of TypeScript projects that want to use type providers. We specify these in the following section. Then we describe the provided import syntax and give an overview of provider options supported by the three implemented type providers (CSV, XML, and JSON-Zod).

To see the type providers in actual use, you can check the example projects in the `examples` directory. There are two applications per each implemented type provider, each of them showcasing a different data processing scenario which is described in the project's `readme.md` file.

4.2.1 Installing type providers

Type providers are distributed as regular NPM packages that can be added to a project using the standard `npm install` command.

If your project uses run-time code from the type provider (e.g. a provided `loadFile` function), you should add the type provider package as a regular dependency and not as a development dependency. This means that you should run `npm install` and not `npm install -save-dev`, and the package reference should appear in the `dependencies` section of `package.json` and not in `devDependencies` (nor `optionalDependencies` and `peerDependencies`). This ensures that when

you distribute your own package containing the provided code, the imports contained in the provided code can be resolved correctly.

Type provider packages implemented as part of this thesis have been published under the `@ts-providers` namespace and can be installed as such:

```
npm install @ts-providers/csv
npm install @ts-providers/xml
npm install @ts-providers/json-zod
```

4.2.2 Provided import syntax

We have presented the provided import syntax throughout the thesis (see e.g., Section 2.3). To reiterate, provided imports extend the syntax of regular ES module imports by adding the `provided` keyword and using the import attributes block after the `with` keyword to pass configuration to the imported type provider package.

For example, the statement in Code Listing 4.3 represents a provided import declaration that imports the `provider` object generated behind the scenes by the `@ts-providers/csv` type provider. The `sample` attribute specifies a relative path to the file that is used for inference of the data shape used to generate the provided types and loader functions.

Code Listing 4.3 Usage of import provided statement

```
import provided { loadFile } from "@ts-providers/csv" with {
    sample: "../sample.csv" }
```

Standard ESM import renaming, as well as the use of TypeScript's namespace imports and the `type` keyword, are supported. Code listing 4.4 illustrates the use of these features. See [19] for more details about ESM imports, and [47] for treatment of imports in TypeScript.

Code Listing 4.4 Supported variants of import provided statement

```
// Aliased named import
import provided { loadFileSync as getCsv } from
    "@ts-providers/csv" with { sample: "../sample.csv" }
const data = getCsv(...);

// Namespace import, CSV includes all provided values and types
import provided * as CSV from "@ts-providers/csv" with { sample:
    "../sample.csv" }
const data = CSV.loadFileSync(...);

// Type-only import for the Row interface
import provided { provider, type Row } from "@ts-providers/csv"
    with { sample: "../sample.csv" }
const data: Row[] = provider.loadFileSync(...);
```

Although it is currently not supported to directly re-export a provided import (i.e. to write `export provided ...`), you can export a provided object in a

separate `export` statement. Such an exported object can be then imported using a regular, non-provided import in other source files or packages, as shown in Code Listing 4.5.

Code Listing 4.5 Export of provided-imported object

```
1 // In 'src/a.ts':
2 import provided { provider } from "@ts-providers/csv" with {
     sample: "../sample.csv" }
3 export  { provider };
4
5 // In 'src/b.ts':
6 import { provider } from "./a";
7 const data = provider.loadFileSync(...);
```

4.2.3 Features of the implemented type providers

In this section, we document the supported options and the provided functionalities of the individual type provider packages implemented for the thesis.

All three packages have the same basic interface for supplying the type inference samples. Specifically, these provided import options can be used:

- `inlineSample <string>` - Specifies the sample data in-code as a constant string literal directly into the import statement. Use `"\n"` to indicate line breaks in the data. String template literals (e.g. `abc`) cannot be used in the provider options block.
- `sample <path>` - Specifies a local file system path to the file containing the sample data. Both absolute and relative paths are supported. The slash (`/`) character should be preferred for directories as it is supported on all platforms. Escaped backslash (\) is also supported on Windows. Text encoding used for reading the file can be optionally specified using the `encoding` provider option. Supported encodings are listed in [48]. The default encoding is UTF-8.
- `httpSample <URL>` - Specifies a URL used to retrieve the sample data with an HTTPS request. Only simple GET requests without custom parameters or headers are currently supported. In particular, this means that the data must be available without authorization.

At least one of these options needs to be set and valid. If multiple options are set in the same import, the method of acquiring the sample is selected in the following order of precedence from highest to lowest: `inlineSample`, `sample`, `httpSample`. If no valid sample is specified, or the type provider encounters an error while loading the sample, an error diagnostic is given to the user, and the provider does not generate or export any code.

CSV provider

The CSV provider attempts to parse the sample data and infer the types of the CSV columns. The inference algorithm is described in Section 3.2.3. The behavior of the provider can be customized using the following options:

- `separator <string>` - Specifies what character (or sequence of characters) is used as a separator when parsing the sample data. The comma (`,`) is used by default.
- `hasHeader (true|false)` - Specifies whether the sample data starts with a header row containing the names of the columns. By default, the provider expects a header row. The behavior can be changed by setting this option to `"false"`.
- `rowTypeName <string>` - By default, the provider generates a TypeScript interface named `Row` to describe the CSV columns. This option can be used to generate an interface with a different name.
- `preserveNames (true|false)` - By default, the provider tries to convert the column names to camel-case (standard style for TypeScript properties). The behavior can be changed by setting this option to `"false"`.

If the type provider parses the sample successfully, it generates a virtual TypeScript module (source file) based on the result of the inference algorithm. This module exports the following objects:

- `parseText(text, separator?): Row[]` – Parses and validates a string input and, on success, produces an array of parsed rows. By default uses the same separator as was used for parsing the sample. This can overridden with the optional function argument.
- `loadFileSync(path, separator?, encoding?): Row[]` - Reads a file using the `readFileSync` function from the `node:fs` library, then parses the content using `parseText`. Apart from the separator, the default encoding can be also changed using the second optional argument.
- `loadFile(path, separator?, encoding?): Promise<Row[]>` – Asynchronous variant of `loadFileSync`.
- `loadHttp(url, separator?): Promise<Row[]>` – Asynchronously loads data using a simple HTTP GET request, then parses the content using `parseText`. The popular `axios` library [49] is used to perform the HTTPS request.
- `loadSampleSync(): Row[]` – Attempts to read and parse data from the same source that was used for the sample. Throws an `Error` if `httpSample` was used.
- `loadSample(): Promise<Row[]>` – Asynchronous variant of `loadSampleSync`. Can be used with `httpSample`.

All of the provided functions perform simple type validation during parsing and throw an `Error` if the actual data does not conform with the inferred column type. If successful, they return an array of type `Row[]` containing the entire data parsed into individual rows with values converted according to the inferred interface (e.g. numeric values converted to `number`). If the sample contained a header (i.e. the columns have names), the run-time parser will look up columns

according to their names instead of just their position. This means that the actual data can include additional columns or have columns in different order. As long as all columns from the sample are in the parsed input and contain compatible values, the data will be parsed successfully. Code Listing 4.6 shows an example of using CSV provider to work with a simple CSV file.

Code Listing 4.6 Example of using CSV provider

```
1  import provided * as people from "@ts-providers/csv" with {
       sample: "../data/people.csv" }
2
3  const printPersonInfo = (person: people.Row): void => {
4      console.log(‘Name: ${person.name}‘);
5      console.log(‘Age: ${person.age}‘);
6      console.log(‘City: ${person.city}‘);
7      console.log(‘Birth Date: ${person.birthDate !== null ?
           person.birthDate : ’N/A’}‘);
8  };
9
10 let people_data = people.loadSampleSync();
11 printPersonInfo(people_data[0]);
```

XML provider

The XML type provider has a very similar user interface as the CSV provider. However, due to the different character of the formats, the inference algorithm (see Section 3.2.4) and the resulting provided type differ significantly. The XML provider generates a set of interfaces describing all the non-leaf elements found in the sample (i.e. elements that have at least one child element or an attribute), as well as a set of functions that can be used to load and parse similar XML data.

Apart from the common sample options described earlier, the provider supports the following import options:

- `preserveNames (true|false)` – By default, the provider tries to convert the element names to Pascal case and the attribute names to camel-case (standard style for TypeScript types and properties, respectively).
- `checkRequired (true|false)` – This option determines if the generated parser code performs validation that all of the properties (representing element attributes and children) that were inferred from the sample as required were found and set while parsing the non-sample data. The behavior is enabled by default.

As mentioned, the XML provider generates source code that exports the element interfaces. It also exports a `provider` object with the same set of functions as the CSV type providers (documented above).

Code Listing 4.7 shows an example of using XML provider to display package references of C# `.csproj` project file.

Code Listing 4.7 Example of using XML provider to display dependencies of .csproj project file

```
import provided { provider } from "@ts-providers/xml" with {
    sample: "../data/csharp_project_file.csproj", checkRequired:
    "false" };

function getPackageNames(path: string): string[] {
    const project = provider.loadFileSync(path);
    return !project.itemGroup ? [] : project.itemGroup
        .filter(group => group.packageReference)
        .flatMap(group => group.packageReference!.map(ref =>
            ref.attributes.include));
}
```

JSON-Zod provider

The JSON-Zod type provider is comparably simpler than the other two packages. It does not use any custom options apart from the common options for samples. Furthermore, it generates source code that exports only two objects:

- `schema` – A prebuilt instance of schema from the Zod validation library [41].
- `SchemaType` – TypeScript type that describes the valid parsed JSON object.

This means that no `provider` object or `load*` functions are generated. Instead, the user can utilize their preferred means of loading the data and then use the built-in functionality of the Zod schema, such as the `parse` and `parseAsync` methods, to parse and validate the JSON data. See the library documentation for more detail [41].

Code listing 4.8 showcases using the type provider to parse and validate data fetched using the popular HTTP client library Axios [49].

Code Listing 4.8 Using JSON-Zod type provider to process HTTP response

```
import provided { schema } from "@ts-providers/json-zod" with {
    sample: "../sample.json" }
import { default as axios } from "axios";

const response = await axios.get("https://...");
const data = await schema.parseAsync(response.data);
```

4.3 Implementing a new type provider package

In this section, we describe the process of implementing a simple type provider package in the form of a short step-by-step tutorial. To follow the tutorial, please set up the required development environment as described in Section 4.1.

Code Listing 4.9 Example of key-value format

```
SOME_KEY=value1
ANOTHER_KEY=value2
...
```

4.3.1 Basic tutorial

As an example, we will implement a provider that infers a TypeScript interface and generates a run-time parser for a simple key-value format similar to the commonly used *.env* files. In the format, each line contains a single key-value pair where the key is separated from the value by a = character. Code listing 4.9 shows an example of the format.

Step 1: Create the project
First, create an empty directory and initialize a Node package by executing `npm init` in an empty directory. During the setup, you can choose any name you want, however, we will assume that the package is named `ts-providers-tutorial--package`. You will later need to use the name when generating the import code for the package. Other initial options are not important.

Add the type provider package dependencies:

```
npm install @ts-providers/compiler @ts-providers/core
```

Then, open the `package.json` file and add a 'main' and 'scripts' sections so that the file looks like in the listing 4.10.

Now when we execute `npm run build` while in the project directory, we invoke the provider-enabled TypeScript compiler. However, first, we need to set up the TypeScript project configuration.

This is done by creating a `tsconfig.json` file in the project root directory. For the purpose of the tutorial, use the settings shown in listing 4.2.

Now we can create a `src/index.ts` source file and verify that the build works by executing `npm run build`. This should emit the `dist/index.js` file, which is configured as the package entry point by the `main` section in `package.json`.

Step 2: Parse the data format
Before we delve into the specifics of implementing the type provider interface, let's first deal with processing our example data format.

Our goal is simple, we want to be able to take a sample string input, parse it according to the rules of the format, and return some representation of the sample's content that will be used to generate the appropriate TypeScript source code that will be provided by our package to the end user.

In this tutorial, we are parsing a list of `KEY=VALUE` pairs and want to provide a TypeScript interface that describes the known keys found in the sample. For simplicity, we will handle the values as `string` and not try to infer more specific types for them. If we omit checking for invalid inputs, this reduces our task to something as simple as the following function:

Code Listing 4.10 Example of a package.json configuration for provider package

```
{
  "name": "ts-providers-tutorial-package",
  "version": "0.0.1",
  "main": "dist/index.js",
  "type": "commonjs",
  "files": [ "dist" ],
  "dependencies": {
    "@ts-providers/compiler": "^5.5.3-providers-v13",
    "@ts-providers/core": "^0.0.66"
  },
  "scripts": {
    "build": "tsc-providers -p ."
  }
}
```

Code Listing 4.11 Example: parsing key-value pairs

```
function getKeysFromSample(input: string): Set<string> {
    const lines = input.split("\n");
    const pairs = lines.map(line => line.split("="));
    const keys = pairs.map(pair => pair[0]);
    return new Set(keys);
}
```

Step 3: Use the provider builder

Creating a valid type provider package involves creating and exporting an object that satisfies a certain interface (as described in Section 3.1.3 and further discussed in Section 4.3.2). However, the easiest way to implement a new type provider is to use the factory function `createBasicTypeProvider` included in the `@ts-providers/core` library.

To begin, add these imports to the `src/index.ts` file:

```
import { BasicTypeProviderFunctions, createBasicTypeProvider }
   from "@ts-providers/core";
```

The `BasicTypeProviderFunctions` interface specifies the necessary functionality that needs to be implemented in the provider package and passed as an argument to the `createBasicTypeProvider` function, which in turn returns the complete type provider instance that we can export from the type provider package for the end users. This gives us the following blueprint for our provider implementation:

Code Listing 4.12 Key-value type provider implementation

```
const providerFuncs:
   BasicTypeProviderFunctions<BasicTypeProviderOptions,
   Set<string>> = {
```

```
    infer: function (textInput: string): Set<string> {
        throw new Error("Function not implemented.");
    },
    generateCode: function (inferenceResult: Set<string>):
       ts.Statement[] {
        throw new Error("Function not implemented.");
    }
}

const provider = createBasicTypeProvider(providerFuncs);
export default provider;
```

Notice that the `BasicTypeProviderFunctions` interface requires us to implement two functions:

1. `infer` – that takes the sample content as a string and returns the inferred data. We already implemented this in the previous step as the `getKeysFromSample` function.
2. `generateCode` – that takes the result of the `infer` function and generates TypeScript code (a list of `Statement` AST nodes).

The `BasicTypeProviderFunctions` interface also takes two generic type parameters:

1. Model type for the options that the end user can pass to our type provider. For now, we can use the built-in `BasicTypeProviderOptions` interface, which contains standard options for specifying the sample. Section 4.3.2 describes how to provide a custom options implementation.
2. Type of the object produced by the `infer` function and consumed by the `generateCode`. Because we already implemented the `getKeysFromSample` function, we know we want to use `Set<string>` here.

Step 4: Implement code generation

What remains is to implement the `generateCode` function, which creates statement AST nodes for the virtual TypeScript source file returned to the end user by the type provider. The most straightforward (and the most performant) way to generate TypeScript AST is to use the official compiler API that can be accessed via the `factory` object.

It should be said that apart from some partial tutorials (see e.g. [34], [50]), there is no comprehensive documentation of the TypeScript AST or the `factory` API. In practice, the best resource for working with the compiler API is a tool such as the TS AST Viewer [51]. This site allows the user to input a piece of TypeScript source code and examine its AST and the `factory` calls required to generate such code.

Let's say we want to provide the user with code such as this:

Code Listing 4.13 Example of the provided code

```
export interface Settings {
  someKey?: string;
  anotherKey?: string;
  // Etc...
}

export const parse = function (input: string): Settings {
  // Here would be code that parses the input
  // and collects the key-value pairs in the Settings record
}
```

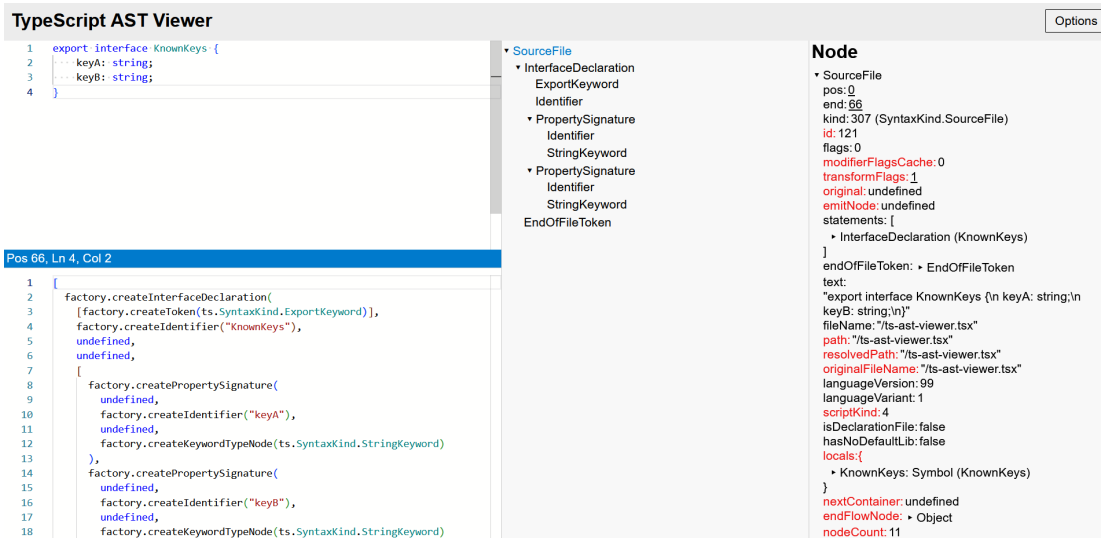

Figure 4.1 Usage of TS AST Viewer tool

We can simply write the code ourselves, insert it into the AST Viewer, and use the outputted factory method calls. We would need to modify the part that generates code for the properties in the `Settings` interface so that it generates properties based on the keys inferred from the sample.

However, the reader would see that with increasing code complexity, the necessary factory calls are quite verbose and quickly become difficult to read and maintain. To alleviate this problem, the `@ts-providers/core` library implements some utility functions and types. For example, the `DeclarationDescriptor` and `TypeDescriptor` classes make generating interface and type declarations easier. With these, we can implement the interface generation like in code listing 4.14.

Code Listing 4.14 Using utility functions to generate an interface declaration

```
// Generates AST for interface declaration with string properties
   for each key found in the sample.
function generateSettingsInterface(keys: Set<string>): Statement {
    const propertyDescriptors = [...keys.values()]
        .map(key => new DeclarationDescriptor(key, new
           SimpleTypeDescriptor("string"), true));

```

```
    const propertySignatures = propertyDescriptors.map(p =>
        p.createSignature());

    return factory.createInterfaceDeclaration(
        [factory.createToken(SyntaxKind.ExportKeyword)],
        "Settings",
        /*typeParameters*/ undefined,
        /*heritageClauses*/ undefined,
        propertySignatures
    );
}
```

When generating AST for the `parse` function, we can reduce the number of necessary AST factory calls with a simple design pattern that utilizes the fact that **functions are first-class objects in JavaScript**. We will not generate the entire function body AST. Instead, we will write and export the actual function `createParseFunction` in our provider package that returns the `parse` function during runtime (i.e. when the provided code is run by the end-user). This means that we only need to generate AST for the following code:

Code Listing 4.15 Provided parse function export

```
import { createParseFunction } from
    "ts-providers-tutorial-package";
export const parse = createParseFunction();
```

Employing a few utility functions from the `factoryHelper` object (from the `@ts-providers/core` library), we can achieve the result in code listing 4.15 above by using the functions shown in listing 4.16.

Code Listing 4.16 Generating parse function export

```
function generateDependencyImport(): Statement {
    return factoryHelper.createNamedImport(
        "ts-providers-tutorial-package", ["createParseFunction"]);
}

function generateParseFunction(): Statement {
    return factoryHelper.createVariableAssignment(
        "parse",
        factory.createCallExpression(
            factory.createIdentifier("createParseFunction"),
            /*typeArguments*/ undefined,
            /*argumentsArray*/ undefined,
        ),
        VariableKeyword.Const,
        /*exported*/ true
    );
}
```

Finally, for this to work, we also need to implement the actual `createParseFunction` function:

Code Listing 4.17 Implementing createParseFunction

```
export function createParseFunction() {
    return (input: string) => {
        const keyValuePairs = input.split("\n")
            .filter(line => line)
            .map(line => line.split("=")) as [string, string?][];

        return keyValuePairs.reduce((obj, [key, value]) => {
            obj[key] = value;
            return obj;
        }, {} as Record<string, string | undefined>);
    };
}
```

Step 5: Test the provider

With parsing and code generation implemented, we can fulfill the `BasicTypeProviderFunctions` interface and get a fully working type provider:

Code Listing 4.18 Implementing BasicTypeProviderFunctions interface

```
const providerFuncs:
   BasicTypeProviderFunctions<BasicTypeProviderOptions,
   Set<string>> = {
    infer: getKeysFromSample,
    generateCode: function (keys: Set<string>): Statement[] {
        return [
            generateDependencyImport(),
            generateSettingsInterface(keys),
            generateParseFunction()
        ]
    }
}

const provider = createBasicTypeProvider(providerFuncs);
export default provider;
```

To test the provider package in your application, do the following:

- You can publish the type provider package on NPM using the `npm publish` command and add the published package to your application using standard `npm install <package name>`.
- Alternatively, install the type provider package as a local reference using `npm install ../path/to/provider-package`. See NPM documentation [52] for more detail. Note that with these local package references, there might be issues with loading types from transitive dependencies, leading to somewhat unreliable language service in the IDE.

Then you can start to use the provided import as shown in Section 4.2.

4.3.2 Further customization

The previous section described a simple scenario when implementing a type provider package. There are various aspects in which you might want to customize your provider, some of which we will discuss in the rest of this chapter.

Custom options and accessing provider context

As we have seen, the `BasicTypeProviderFunctions` interface used by the `createBasicTypeProvider` factory function has a generic type parameter for the type that describes options supported by the type provider. These are the options that the end user can pass in the `with` block of the provided import like this:

```
import provided { ... } from "@ts-providers/xml" with { sample:
    "...", preserveNames: true }
```

To specify custom options of a provider (created by `createBasicTypeProvider` factory function), you must declare an interface that extends the `BasicTypeProviderOptions` interface and then use this extended interface as the first type parameter of `BasicTypeProviderFunctions` interface. When you do this, you will need to implement an additional function `parseCustomOptions` as part of the `BasicTypeProviderFunctions` interface. There you:

- handle parsing of the optional string-typed options passed from the end user,
- specify their default values.

Code Listing 4.19 illustrates this process.

Code Listing 4.19 Customizing type provider options

```
interface MyOptions extends BasicTypeProviderOptions {
  preserveNames: boolean;
}

const providerFuncs: Basic\-Type\-Provider\-Functions<MyOptions,
    T> = {
  parseCustomOptions: (unparsedOptions) => {
    return {
      preserveNames: unparsedOptions.preserveNames === "true" ?
          true : false
    }
  }, ...
}
```

You can access the parsed options (as well as the standard ones such as `sample`) via the second argument that is passed to your `infer` and `generateCode` functions (see Code Listing 4.20).

Code Listing 4.20 Accessing type provide options

```
const providerFuncs: Basic\-Type\-Provider\-Functions<MyOptions,
    T> = {
  infer: (input, options: MyOptions) => {
    if (options.preserveNames) ...
  }, ...
}
```

Furthermore, you can access a `ProviderContext` instance passed in the third argument. This object contains information from the compiler, namely:

- the `importingFilePath` property that contains the path to the source file with the provided import, which is being currently resolved,
- the `runtimeTarget` property that may contain an identifier of the intended runtime environment that the end user can optionally set in their `tsconfig.json`, e.g. `"node"` or `"browser"`. You can use this information to customize your code generation (as demonstrated in Code Listing 4.21). For instance, you can use different APIs for reading files or altogether skip generating certain functions.

Code Listing 4.21 Accessing runtimeTarget property

```
const providerFuncs: Basic\-Type\-Provider\-Functions<MyOptions,
    T> = {
  generateCode: (inferenceResult, options, context) => {
    switch (context.runtimeTarget) ...
  }, ...
}
```

Emitting custom diagnostics

The standard type provider created by `createBasicTypeProvider` has a set of diagnostics that get returned to the compiler in case of various common issues and are displayed either in the standard output (during command-line compilation) or as "squiggles" in the IDE. These cover problems such as the provider not being able to read the specified sample file.

You can emit custom diagnostics using the `DiagnosticCollector` instance given as the last argument to your provider functions. This object exposed functions `addGeneralDiagnostic` and `addOptionDiagnostic`. The former is for messages reported as issues with the entire provided import, while the latter lets you report issues with a particular provided import option. Both take a message in the format specified by the `DiagnosticMessage` interface from the TypeScript compiler API.

To indicate that the provider pipeline should not continue and return a failed result (with all the diagnostics added so far), you can return `undefined` instead of the regular result from any of the basic provider functions as demonstrated in Code Listing 4.22.

Code Listing 4.22 Returning undefined to indicate that type provider should not generate any code due to error

```
const providerFuncs: Basic\-Type\-Provider\-Functions<MyOptions,
    T> = {
  parseCustomOptions: (unparsedOptions, context, diagnostics) => {
    if (unparsedOptions.someOpt && unparsedOption.anotherOpt) {
      diagnostics.addOptionDiagnostic("anotherOpt",
          Messages.ConflictingOptions);
      return undefined;
    }
  }, ...
}
```

Custom providers without `createBasicTypeProvider`

Implementing provider packages with the `createBasicTypeProvider` factory function should cover the most common scenarios. However, if more flexibility is required, you can implement the provider from scratch. As described in Section 3.1.3, the only requirement is to implement at least one of the interfaces `SyncTypeProvider<TOptions>` or `AsyncTypeProvider<TOptions>`. See the implementation in `providers/core/src/provider-utils/provider-builder.ts` for reference.

5. Conclusion

The goal of this thesis was to analyze and implement support for type providers in TypeScript.

We have analyzed various design, usability, and implementation-related aspects of the problem in Chapter 2. All of the requirements prescribed for the solution were met. We successfully modified the TypeScript compiler to support compile-time invocation of type provider packages based on a new provided import syntax as described in Section 3.1. The invoked package is able to provide the compiler with types based on inference from data samples or other compile-time computation.

The compiler is then able to use the provided types during standard type checking and to enable code editor features such as auto completions and reporting type errors with compiler diagnostics. The provider can also generate validating data access code that enforces that run-time types do not violate the static type guarantees, thus improving the safety of TypeScript programs without the need to write additional code by hand or involve external code generation tools.

The viability of the solution has been demonstrated by implementing type provider packages for JSON, XML, and CSV (Section 3.2). Despite the prototype character of the implementation, the showcase packages can already be used in practical scenarios. Additional type providers can be implemented without further modifications to the compiler.

5.1 Future work

There are many directions for further development of the prototype solution we presented in this thesis. We outline some of them here.

Better management of provided code

The prototype implementation does not support incremental parsing, which is one of the important optimizations in the language service. Even though the main overhead of loading samples and running the type inference is mitigated by caching the inference results, the provided AST is still regenerated too often.

However, the incremental parsing is disabled for the provided code as its current implementation is closely tied to the textual base of the processed source files loaded from disk. Implementing a dedicated cache for provided source files might be a better solution than rewriting the incremental parser subsystem. Virtualizing the file system for provided source files akin to [53] might also be an option. This might allow for a more robust handling of provided source files to be implemented in the module loader so that the solution does not depend on generating unique file names.

Lazy evaluation

Implementing support for providing types lazily, as was done in F# would open possibilities to use type providers as tools for working with complex information spaces.

Such a feature would most likely require significant changes to the implemented solution. A possible method would be to augment the generated AST nodes (and possibly the Type records created from them) with type provider-specific metadata, which would indicate that this node can possibly be further "extended". Then, during certain operations, such as when the language service asks the checker for a list of members in order to populate the auto-completion box, the compiler would query the type provider again and request providing the members of said type.

Metaprogramming

Adding metaprogramming capabilities to type providers seems possible without requiring extensive changes to our solution. The provider could be given easier access to the importing source file or the entire set of source files processed by the compiler. Then, the provider could generate code based on the analysis of other source code rather than data samples.

Typed import options

An immediate improvement in usability of type providers would be to implement proper type checking of import options that are specified in the import attributes block. Currently, the options are only checked manually in the respective type provider implementation, which can return diagnostics targeted at the specific import options. A better solution would be to extend the provider invocation API through which the compiler could request the options interface from the type provider. Then, it could type check the import attribute block as an object instance of such interface. Implemented properly, this could "for free" add support for auto completions and standard compiler emitted diagnostics.

Multiple samples

Currently, there is no clean way to specify multiple samples for the provider to use (other than e.g. to concatenate a list of paths into a single string). This makes it less viable to depend on sample engineering in real-world projects.

Bibliography

1. Chiusano, Paul. *The advantages of static typing, simply stated* [https://pchiusano.github.io/2016-09-15/static-vs-dynamic.html]. [N.d.]. [cit. 2024-07-16].
2. Pawar, Vaibhav. *What are the advantages and disadvantages of the database first approach in .net core (EF core)?* [https://medium.com/@vaibhavpwr/what-are-the-advantages-and-disadvantages-of-the-database-first-approach-in-net-core-ef-co-a0dc1a01b5ad]. 2023. [cit. 2024-07-16].
3. *Code generation and version control* [https://www.jooq.org/doc/latest/manual/code-generation/codegen-version-control/]. [N.d.]. [cit. 2024-07-16].
4. Syme, Don; Battocchi, Keith; Takeda, Kenji; Malayeri, Donna; Fisher, Jomo; Hu, Jack; Liu, Tao; McNamara, Brian; Quirk, Daniel; Taveggia, Matteo; Chae, Wonseok; Matsveyeu, Uladzimir; Petricek, Tomas; and. *F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources*. Microsoft Research, 2012-09. Tech. rep., MSR-TR-2012-101. Available also from: https://www.microsoft.com/en-us/research/publication/f3-0-strongly-typed-language-support-for-internet-scale-information-sources/.
5. Syme, Donald; Battocchi, Keith; Takeda, Kenji; Malayeri, Donna; Petricek, Tomas. Themes in information-rich functional programming for internet-scale data sources. In: *Proceedings of the 2013 Workshop on Data Driven Functional Programming*. Rome, Italy: Association for Computing Machinery, 2013, pp. 1–4. DDFP '13. isbn 9781450318716. Available from doi: 10.1145/2429376.2429378.
6. Petricek, Tomas; Guerra, Gustavo; Syme, Don. Types from data: making structured data first-class citizens in F#. In: Krintz, Chandra; Berger, Emery D. (eds.). *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 2016, pp. 477–490. Available from doi: 10.1145/2908080.2908115.
7. Christiansen, David. Dependent type providers. In: 2013, pp. 25–34. Available from doi: 10.1145/2502488.2502495.
8. *Scala docs – macros, type providers* [https://docs.scala-lang.org/overviews/macros/typeproviders.html]. [N.d.]. [cit. 2024-07-16].
9. *Scala docs – macros, blackbox vs whitebox* [https://docs.scala-lang.org/overviews/macros/blackbox-whitebox.html]. [N.d.]. [cit. 2024-07-16].
10. *JetBrains – Developer Ecosystem survey 2023* [https://www.jetbrains.com/lp/devecosystem-2023/languages/]. [N.d.]. [cit. 2024-07-16].
11. *Typescript Transpiler Explained* [https://daily.dev/blog/typescript-transpiler-explained]. [N.d.]. [cit. 2024-07-16].

12. *GitHub – TypeScript, issue #3136 – Feature Request: F# style Type Provider support?* [https://github.com/microsoft/TypeScript/issues/3136]. [N.d.]. [cit. 2024-07-16].
13. *GitHub – TypeScript, issue #16607 – Allow "Compiler Plugins"* [https://github.com/microsoft/TypeScript/issues/16607]. [N.d.]. [cit. 2024-07-16].
14. *GitHub – TypeScript, issue #14419 – Plugin Support for Custom Transformers* [https://github.com/microsoft/TypeScript/issues/14419]. [N.d.]. [cit. 2024-07-16].
15. *GitHub – TypeScript Design Goals* [https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals]. [N.d.]. [cit. 2024-07-16].
16. *GitHub – ts-creator* [https://github.com/HearTao/ts-creator]. [N.d.]. [cit. 2024-07-16].
17. *Effective TypeScript – The Seven Sources of Unsoundness in TypeScript* [https://effectivetypescript.com/2021/05/06/unsoundness/]. 2021. [cit. 2024-07-16].
18. *TypeScript Handbook – Modules* [https://www.typescriptlang.org/docs/handbook/2/modules.html]. [N.d.]. [cit. 2024-07-16].
19. *mdn web docs – import* [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import]. [N.d.]. [cit. 2024-07-16].
20. *GitHub – TypeScript Using the Language Service API* [https://github.com/microsoft/TypeScript/wiki/Using-the-Language-Service-API]. [N.d.]. [cit. 2024-07-16].
21. *JetBrains Blog – JavaScript and TypeScript Trends 2024: Insights From the Developer Ecosystem Survey* [https://blog.jetbrains.com/webstorm/2024/02/js-and-ts-trends-2024/]. [N.d.]. [cit. 2024-07-16].
22. *GitHub – ts-morph* [https://github.com/dsherret/ts-morph]. [N.d.]. [cit. 2024-07-16].
23. *webpack* [https://webpack.js.org/]. [N.d.]. [cit. 2024-07-16].
24. *rollup.js* [https://rollupjs.org/]. [N.d.]. [cit. 2024-07-16].
25. *GitHub – Writing a Language Service Plugin* [https://github.com/microsoft/TypeScript/wiki/Writing-a-Language-Service-Plugin]. [N.d.]. [cit. 2024-07-18].
26. *GitHub – TypeScript* [https://github.com/microsoft/TypeScript]. [N.d.]. [cit. 2024-07-18].
27. *TypeScript Deep Dive – TypeScript Compiler Internals* [https://basarat.gitbook.io/typescript/overview]. [N.d.]. [cit. 2024-07-18].
28. *Anders Hejlsberg on Modern Compiler Construction* [https://learn.microsoft.com/en-us/shows/seth-juarez/anders-hejlsberg-on-modern-compiler-construction]. [N.d.]. [cit. 2024-07-18].
29. *GitHub – TypeScript –checker.ts* [https://github.com/microsoft/TypeScript/blob/main/src/compiler/checker.ts]. [N.d.]. [cit. 2024-07-18].

30. *mdn web docs – The event loop* [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop]. [N.d.]. [cit. 2024-07-18].

31. Creager, Joe. *5 Reasons to Avoid Deasync for Node.js* [https://joecreager.com/5-reasons-to-avoid-deasync-for-node-js/]. [N.d.]. [cit. 2024-07-18].

32. *Node.js v22.4.1 documentation – Worker threads* [https://nodejs.org/api/worker_threads.html]. [N.d.]. [cit. 2024-07-18].

33. *mdn web docs – Atomics.wait* [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics/wait]. [N.d.]. [cit. 2024-07-18].

34. *GitHub – TypeScript-Website – Using the Compiler API* [https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API]. [N.d.]. [cit. 2024-07-18].

35. *CSV, Comma Separated Values (RFC 4180)* [https://www.loc.gov/preservation/digital/formats/fdd/fdd000323.shtml]. [N.d.]. [cit. 2024-07-18].

36. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [https://www.w3.org/TR/xml/]. [N.d.]. [cit. 2024-07-17].

37. *XML Schema Tutorial* [https://www.w3schools.com/xml/schema_intro.asp]. [N.d.]. [cit. 2024-07-17].

38. *xmldoc library* [https://www.npmjs.com/package/xmldoc]. [N.d.]. [cit. 2024-07-17].

39. *JSON format* [https://www.json.org/json-en.html]. [N.d.]. [cit. 2024-07-17].

40. *Ajv JSON schema validator* [https://ajv.js.org/]. [N.d.]. [cit. 2024-07-17].

41. *Zod schema validation* [https://zod.dev/]. [N.d.]. [cit. 2024-07-16].

42. *TypeScript Handbook – tsc CLI Options* [https://www.typescriptlang.org/docs/handbook/compiler-options.html]. [N.d.]. [cit. 2024-07-16].

43. *npm Docs – package.json* [https://docs.npmjs.com/cli/v10/configuring-npm/package-json]. [N.d.]. [cit. 2024-07-16].

44. *TypeScript – Compiler Options* [https://www.typescriptlang.org/tsconfig/]. [N.d.]. [cit. 2024-07-16].

45. *Bun.js* [https://bun.sh/]. [N.d.]. [cit. 2024-07-16].

46. *Visual Studio Code* [https://code.visualstudio.com/]. [N.d.]. [cit. 2024-07-16].

47. *TypeScript – Modules – Theory* [https://www.typescriptlang.org/docs/handbook/modules/theory.html]. [N.d.]. [cit. 2024-07-18].

48. *Node.js v22.4.1 documentation – Buffer* [https://nodejs.org/api/buffer.html]. [N.d.]. [cit. 2024-07-16].

49. *Axios* [https://axios-http.com/]. [N.d.]. [cit. 2024-07-16].

50. *Gentle Introduction To Typescript Compiler API* [https://january.sh/posts/gentle-introduction-to-typescript-compiler-api]. [N.d.]. [cit. 2024-07-18].

51. *TypeScript AST Viewer* [https://ts-ast-viewer.com/]. [N.d.]. [cit. 2024-07-16].

52. *npm-install Docs* [https://docs.npmjs.com/cli/v8/commands/npm-install]. [N.d.]. [cit. 2024-07-16].

53. *GitHub – TypeScript-Website – VFS* [https://github.com/microsoft/TypeScript-Website/tree/v2/packages/typescript-vfs]. [N.d.]. [cit. 2024-07-18].

A. Attachments

A.1 Overview of attached files

.
- **compiler-diff.html** - Visualization of git diff output showing our modifications to the TS compiler codebase
- **readme.md** - Overview of attached files
- **compiler** - Contains source code of the modified TS compiler
- **examples** - - Contains source code of demo applications that showcase use of type providers. See Section A.2 for descriptions of individual demos
- **providers** - Contains source code of the implemented type provider packages
  - **core** - Support library for type provider implementation (@ts-providers/core)
  - **csv** - The CSV type provider package (@ts-providers/csv)
  - **json-zod** - The Json-Zod type provider package (@ts-providers/json)
  - **xml** - The XML type provider package (@ts-providers/xml)

A.2 Example applications

Please, follow the instructions in Chapter 4 to set up your environment. Run `npm install` in each example directory and select your TypeScript version in VS Code.

CSV people
This example shows the usage of the CSV provider on a CSV file with varying data types of columns. The example demonstrates that the type inference algorithm for CSV can correctly infer the most specific type applicable for each column, taking nullability into account.

It also demonstrates that once the column types of the CSV file are inferred, an attempt to read a differently typed CSV file via the given CSV provider instance will raise a validation error.

CSV cityvizor
This example demonstrates the usage of the CSV provider on real-world data format used by an open-source accounting application, Cityvizor `https://cityvizor.cz/landing/`.

The example reads the accounting data of a municipality and computes its overall balance.

JSON weather
This example uses the JSON-zod provider to infer a schema of HTTP responses of

API endpoints. That provides out-of-box validation of received HTTP responses without manually implementing data models or validation code on a user's part.

The example provides a simple console application that, given the name of a city and the number of days, will return an average temperature in the given city for a given number of days from now.

Usage:

```
npm install
npm run build
cd dist
node index.js -d <numDays> <cityName>
```

JSON postman
Postman is a popular tool among web developers as it enables them to easily test web APIs. Postman allows to create collections of HTTP requests that can then be exported in JSON format to be stored, shared, and reused. This example uses JSON type provider to process the postman collection of HTTP Cats API `https://http.cat/`.

XML .csproj
One of the use cases of XML format is C# `.csproj` project files. These files define the project's dependencies on other projects and packages.

This example uses XML provider to print all package references of all '.csproj' files located under a given directory.

Usage:

```
npm install
nmp run build
cd dist
node index.js <directoryPath>
```

XML HTML A very common usage of XML format is HTML. This example demonstrates that XML type provider can be used to implement an HTML scraper to extract data from HTML easily.