Employment of TypeScript's typing capabilities to develop a spreadsheet editor software

Gouveia, Andreia
up201706430@edu.fe.up.pt

Cardoso, João
up201705149@edu.fe.up.pt

Lajes, Sofia
up201704066@edu.fe.up.pt

***Abstract*— Programming languages with weak typing dynamics - such as JavaScript - often fall into issues that are extremely hard to solve because they are usually only discovered during the run-time phase. Languages with strong typing capabilities - such as TypeScript - always point out the compilation errors at the time of development. Because of this, in the run-time, the chance of getting errors is much smaller. In order to better understand the perks and flaws of the these languages, the group decided to develop a spreadsheet editor web-app, which code strongly relies on typing and the dynamics between different types and type creation. By doing so it was possible to observe the impact that strong typing mechanisms have on code creation and error reduction, therefore, contributing to cleaner, more robust coding.**

I. INTRODUCTION

Today there is an enormous variety of programming languages, that differ in all sorts of categories, such as the syntactic and semantic ruling, which have subsequent effects on the strictness level of typing rules.

Programming languages with weak typing dynamics - such as JavaScript - introduce perks such as compilation and run-time speed, code simplicity, interoperability, and versatility. However, they often fall into issues that are extremely hard to solve, notably the lack of debugging facility - since compilation errors are usually only discovered during run-time.

On the other hand, strongly typed languages have stricter typing rules at compile time, and, therefore, errors and exceptions are prone to happen during the compilation phase (such as variable assignment, procedure arguments and function calling). Consequently, the chance of getting errors much smaller, whereas JavaScript is an interpretive language.

This project's code strongly relies on typing and the dynamics between different types and type creation. By doing so it was possible to observe the impact that strong typing mechanisms have on code creation and error reduction, therefore, contributing to cleaner, more robust coding.

Ultimately, the goal was to develop a domain specific language, defining appropriate syntax and semantics, and heavily resorting to string typing.

II. PRODUCT

In order to better explore the concept of typing mechanisms and its effect on code development, compilation and run time speed and debug, the group elaborated a spreadsheet editor using Typescript. For the front-end we used ReactJS, and for the back-end we used NodeJS. These components communicate with each other using HTTP requests. The web-app has the expected basic functionalities of a spreadsheet editor, such as numbers, strings, references to other cells and the respective formulas to manipulate these values. It innovates by permitting declarations and operations with more more complex data types: arrays of numbers and JSON-like objects.

III. CHALLENGES AND SOLUTIONS

A. Types

The first step in the implementation stage was defining the types of values the program would support. The group settled on 4 basic types: numbers, strings, arrays and objects. Typescript allows the creation of Union Types, which were used to make things easier:

- Literal = number | string | number[] | ObjectLiteral - representation of the basic types of values a cell can hold;

- ObjectLiteral = Record⟨string, number | string | number[]⟩- objects are a relation between string keys to values of all literal types, except ObjectLiterals themselves.

B. Cell representation

To create a spreadsheet editor, the cells need to be represented in a way that is easy to define different types of cells. Achieving this was possible by following an Object Oriented Programming paradigm: a simple Abstract class Cell was implemented, with information regarding it's coordinates on the grid and two abstract methods to return the value the cell contains:

- getValue() - this method returns a Literal. Its function is to calculate the value of the cell, to use in other operations and/or to show the user.

- content() - this method returns a string representing the input given by the user in that cell. This is especially useful to show the formulas the user typed in that cell.

Various child classed of the class Cell were also implemented, these being separated in two categories:

- literal cells - cells that represent all the basic literals, like numbers, strings, arrays and objects.

- formula cells - cells that represent all formulas, like SUM (addition), SUB (subtraction), MUL (multiplication), etc...

C. Cell referencing and circular dependencies

Another concern when developing a spreadsheet editor is how to implement cell referencing, that is, the ability to reference values in other cells. An effective implementation requires a way to update the values of all the cells that depend on a specific cell every time this last cell changes its value.

To solve this problem, a graph-like approach was used. Every cell has a list of dependents, that is, all the cells that depend on it. When its value is updated, the cell goes through the list of dependents, triggering a recalculation of the cells' values. This is done recursively, to ensure every single cell that needs its value changed in properly updated.

This recursive solution however gives rise to another concern: circular dependencies. If they were present, the program would be stuck in an infinite loop, so they need to be detected and prohibited. This is done when creating a Formula Cell, as these are the only types of cells that can have references to other cells. First, the base case: in case a cell with a formula references itself directly, a circular dependency is registered and an error is presented to the user. Second, indirect self-references are detected by looking at the dependents of the cell: if a cell, A, that is being referenced by the formula, B, is a dependent (direct or indirect) of the formula itself (A $\rightarrow$ B, meaning the value of A is dependent of the value of B), then a circular dependency would happen (B $\rightarrow$ A $\rightarrow$ B). Because the indirect dependents are being used, this logic works for an infinitely big chain of referencing cells.

D. Grammar extensibility

The domain specific language was developed with the Parsimmon package, allowing the use of parser combinators to easily define and implement a language for the spreadsheet. Because multiple operations had to be implemented, there was a need to make the code as easy to expand and extend as possible. The solution proposed to achieve this was, instead of creating a grammar rule for every single operation, to create a rule for all operations. Syntactically, the name of the operation ("SUM"/"SUB"/etc...) is the first thing to parse, followed by the token "(", a list of zero or more operands, separated by a ",", followed by a ")".

To make more operations easier to add, a function was created that returned an array of Parsers, one for every operation name. This alone would not be enough to make the grammar easily extensible, as the correct class still needed to be instantiated depending on what operation was being parsed. So an object literal, Operations, was defined that mapped the string of an operation name with its corresponding class where the functionality of that operation was being implemented (Fig. 1).

```
export const Operations = {
    'SUBSTR': SubstrCell,
    'SUM': SumCell,
    'SUB': SubCell,
    'MUL': MulCell,
    'DIV': DivCell,
    'LEN': LenCell,
    'AT': AtCell,
    'PROP': PropCell,
    'MAX': MaxCell,
    'MIN': MinCell,
    'ARRAY': ArrayOpCell,
    'CONCAT': ConcatCell,
    'AVRG': AvrgCell
}
```

So now, the class that needs to be instantiated is the value in Operations mapped to the key of the string that was parsed.

IV. CONCLUSIONS

With this project, the group was able to experience the advantages and disadvantages of programming languages with strong typing, as well as how to make robust, modular, extensible code. Strong typing required more focus on previous planning; as development grew, the group realized that there were better ways of implementing problems that had been previously resolved (for example, improving modularity and reusability in functions by creating the type "Literal", mentioned in the previous section). This implies that, in order to make it the right way, due to all this nuances, it took longer to define the language and architecture. In this situation, weak typing languages are more accessible. However, the first ensured the program was prepared to deal with typing incompatibilities or misuse.

Overall, since the product needed the concept of types in order to define ruling - such as what types can be used in each operation - it was nearly unthinkable to do this on a weak typing language. In this case, strong typing excels over weak typing.

The group believes that the goals of the subject were achieved, and the result exceeded initially determined expectations.

REFERENCES

[1] BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. Understanding typescript. In: European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 2014. p. 257-281.

[2] RASTOGI, Aseem, et al. Safe efficient gradual typing for TypeScript. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2015. p. 167-180.