UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2568

Design of a strongly-typed programming language

Petar Mihalj

Zagreb, May 2021.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 12 March 2021

MASTER THESIS ASSIGNMENT No. 2568

Student:	**Petar Mihalj (0036497900)**
Study:	Computing
Profile:	Computer Science
Mentor:	asst. prof. Ante Đerek

Title: **Design of a strongly-typed programming language**

Description:

Programming languages play a central role in the development of software. However, there is no silver bullet programming language, which would be efficient, easy to use, portable and consistent. The goal of this graduate thesis is to design a new programming language with the emphasis on how strong typing can address common drawbacks. The language specification should be described in a formal form. Design should be followed-up by a careful analysis of some aspects of the language, for example to prove the desirable properties of the type system. The final goal of the thesis is to implement the tool chain, which primarily consists of a compiler, but can include other tools, such as language servers and formatters. The thesis should be accompanied with the source code of all developed software. All references should be clearly cited. Any assistance received should be clearly acknowledged.

Submission date: 28 June 2021

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2568

Pristupnik:	**Petar Mihalj (0036497900)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Ante Đerek

Zadatak:	**Oblikovanje strogo tipiziranog programskog jezika**

Opis zadatka:

Programski jezici centralna su karika u razvoju programske potpore. Ipak, ne postoji programski jezik najbolji za sve primjene, u isto vrijeme efikasan, jednostavan, prenosiv i konzistentan. Cilj ovog diplomskog rada dizajnirati je novi programski jezik, s naglaskom na prednosti strogih tipova u rješavanju čestih problema. Specifikacija jezika treba biti opisana u formalnom obliku. Nakon dizajna jezika potrebno je provesti pažljivu analizu nekih svojstava jezika, na primjer pokazati poželjna svojstva sustava tipova. Posljednji cilj rada implementacija je skupa alata, kojeg primarno čini prevoditelj, ali koji može uključivati i druge alate, poput jezičnih poslužitelja (language servers) i formatera koda. Rad treba uključivati izvorni kod razvijenih alata. Sve reference trebaju biti jasno citirane. Sva primljena pomoć treba biti jasno obznanjena.

Rok za predaju rada: 28. lipnja 2021.

I thank everybody...

CONTENTS

1. Introduction

The programming language developed as a part of this thesis is called AGT. The name AGT stands for an unfortunate fact that the names of all precious stones are taken by other programming languages, hence "All Gems are Taken".

AGT is a statically and strongly typed language, with a highly expressive type system. The type system is used for three main purposes:

1. types determine the size of values in memory
2. types enable polymorphic behaviour (of both builtin and user-defined (struct) types)
3. types allow for compile-time computation

The language also allows for implementation of object lifetime constructs, such as those seen in languages like C++ or Rust; object creation, copying and destruction. These constructs can be used to implement various object ownership semantics, which will be demonstrated in later chapters.

The reference AGT compiler (AGTC) produces executable binaries only. This is in sharp contrast to some other languages, which can produce object files, to later be linked into executables for a particular runtime, or used to augument the runtime environment itself (kernel modules, for example). This decision was made primarely because the goal of the thesis was to study the language from the application programmers perspective. This goal implies that AGT is running on a rich runtime environement, where heap memory management and standard I/O are available. Future updates to AGT specification are meant to allow AGT to be compiled into linkable files compatible with various platform specific ABIs.

The compiler frontend is implemented using Python3 programming language, while the backend uses LLVM (Lattner und Adve, 2004) compiler infrastructure. Allowing for some simplification, AGT is first compiled into LLVM intermediate representation

(LLVM-IR), and than is turned into executable by a chain of tools for compiling LLVM-IR and linking the resultuing object files into executables.

1.1. Paper organization

This master thesis will gradually introduce the reader to AGT, starting from the simplified introduction to syntax and semantics, with an emphasis on type system and inference. This introduction will be augumented with simple examples of AGT code.

After the surface-level exploration of the language, we will study the language in depth. This part will have many references to the AGTC reference implementation, rather than to a separate formal specification. Since AGT is still not a finalized product, formal specification does not exist, and it's functionality is determined by a reference compiler. We will mostly focus on the type system, since it is unorthodox and is the main contribution of the author. In this part, the author will justify the decisions which have been made on all levels of AGT design process.

In the last part, we will address the issues AGT has both on definition and implementation levels, and comment on the various features that can be improved or added to the language.

2. The AGT Programming Language - a gentle introduction

Let us first check out an example AGT program:

```
fn outnl(i){
    out(i);
    out('\n');
}

fn mul(a, b) -> a{
    return a*b;
}

fn main()->i32{
    let a = in<i32>();
    let b = in<i8>();

    let c = mul(a, cast<i32>(b));
    outnl(a);
    outnl(b);
    outnl(c);
}
```

Every AGT program has to have a function called main, which returns a 32-bit-wide integer. Some functions, like `outnl`, don't return anything. It is important to note that you can't specify that a function returns `void` like in C for example.

A `let <x> = <y>;` construct is *initialization assignment* statement. This roughly translates to an allocation of memory on a stack (which is allocated at the newly generated *location* of <x>), and copying the value of expression <y> to the location of identifier <x>.

Note that call of the function `in` is *parametrized* by a *type argument* `i32`. Type arguments are in contrast to the *value arguments*, which carry value too, along with a type. The function `in` is a builtin, and the `i32` is used to signal the compiler that we want an instance of this function which returns an `i32`, rather than a `char`, for example. Of course, user-defined functions can also have *type parameters*, along with *value parameters*; we will get to these later.

Notice that we used the `cast<i32>` builtin function to convert `b` to `i32` before passing it to function `mul`. If we didn't do that, the compiler would signal that it can't compute the expression `a*b`, since multiplication is predefined only for integer types of same size. Apart from the `i8` and `i32` we used in this example, `i16` and `i32` are also available.

There is a certain feature that you might have missed while inspecting the code; namely the abscence of types in the definition of function `mul`. While some `dynamically typed` languages (ex. Python) allow for object of any type to be passsed to the function, AGT behaves quite differently.

Every time AGTC (AGT compiler) encounters a function call, it tries to infer a *function type*. Function type is inferred from function definitions. Consequently, you can think of the definitions in source code more as `templates` for synthesis of actual code, than the actual representation of code. These definitions lack types, and can't be considered in isolation. In this example, the call of function `mul` causes the compiler to try to infer a function type.

It is important to emphasize that one function definition can be a source of many function types. The `outnl` (output with newline) function is a prime example of this behaviour. It is called three times, with argument of types `i32`, `i8` and `i32`. Thus, the compiler has to infer two function types, one which can be fed with an `i8`, and one with `i32`. The success of this inference corresponds with compilers ability to infer function body of the given function definition. This will be possible if and only if the compiler can also infer the function type `out` function which can take ... The property of `outnl` function definition which allows it to be a source for inference of two distinct function types, given that it *makes sense* for these types (regarding the function body), is called **parametric polymorphism.**

3. Conclusion

Zaključak.

BIBLIOGRAPHY

[Lattner und Adve 2004] **LATTNER, Chris ; ADVE, Vikram:** ***LLVM: A compilation framework for lifelong program analysis transformation*. 2004**

Design of a strongly-typed programming language

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Title

Abstract

Abstract.

Keywords: Keywords.