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".

The AGT programming language 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 memory footprint of values
2. types allow for polymorphic behaviour of operators and functions on all *concrete types* (built-in or struct types)
3. type system lets the programmer perform compile-time computation

The language also allows for implementation of object lifetime constructs, such as those seen in languages as C++ or Rust; object creation, copying and destruction.

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 programmer's 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 compiler infrastructure (Lattner und Adve, 2004). 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 resulting object files into executables.

1.1. Example program

Let us first check out an example AGT program in listing 1.1:

Listing 1.1: Example AGT program

```
1  fn outnl(i){
2      out(i);
3      out('\n');
4  }
5
6  fn mul(a, b) -> a{
7      return a*b;
8  }
9
10 fn main()->i32{
11     let a = in<i32>();
12     let b = in<i8>();
13
14     let c = mul(a, cast<i32>(b));
15     outnl(a);
16     outnl(b);
17     outnl(c);
18 }
```

Every AGT program has to have a function definition named main, which returns a 32-bit-wide integer, and has no parameters. 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 an *initialization assignment statement*. This roughly translates to an allocation of memory on the stack (which can be refered to as the *location* of <x> from now on), and copying the value of expression <y> to the location of identifier <x>.

Note that the function call `in` (line 11) is *parametrized* by a *type argument* `i32`. Type arguments are used to pass types, but not values. They are different than *value arguments*, which carry value too, along with a type. Note that *passing* of type parameters is purely a compile-time concept; types can't be refered to during runtime.

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`, instead of a `bool`, 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 `i64` are also available.

Also, line 6 is terminated by a `-> a`. This is a return type specification; it states that this function will return whatever is type of a.

You might have noticed a certain feature while inspecting the code; namely the abscence of explicitly stated types in the signature of function definition `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 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.

Parameters of this `mul` function definition (a, b) can be fit with objects of any type; it is after this fitting that the compiler will determine that passed types can or can't be used (due to their incompatibility with a function definition body, for example).

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 functionality of body of the given function definition.

In this case, inference of function definition body will be possible if both function calls on lines 2 and 3 can be inferred. Since `out` function is a built-in for both i32 and i8, line 2 won't be problematic for inference. Line 3 will be inferred correctly if AGTC suceeds to resolve function `out` which takes in an argument of type `char`. Since this is also a built-in, inference of function types for both version of `mul` suceeds.

The property of AGT **function calls** which can make them refer to different function types, when called with different sets of arguments, is called *polymorphism*. Some sources label polymorphic the very **function definitions** which, when called upon, can result in such behaviour. We will use these conventions interchangably, since the definition itself is non-rigorous.

Function definition of `outnl` introduces a special type of polymorphism, *parametric polymorphism*. This polymorphic behaviour is said to be present if the function definition can be a source for potentially unlimited number of types, as long as those types are compatible with function definition body.

Python programming language exibits a superficially similar behaviour, but it's function body compatibility checks occur at runtime. The programming style which utilizes this language capability is often called "Duck Typing" (Fred L. Drake Jr.).

1.2. Paper organization

This master thesis will gradually introduce the reader to AGT.

Chapter 1 has already introduced the reader to basic syntax and semantics, using a simple example and emphasizing the surface-level properties of type system and inference.

After the surface-level exploration of the language, chapter 2 will provide an in-depth description of AGT. This chapter 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. Author will justify the decisions which had been made on all levels of AGT design process.

Chapter 3 will supplement the reader with plenty of examples of AGT code. These will further deepen reader's understanding of AGT. The examples describe impementation of common programming paradigms, constructs, patterns in AGT, with a detailed explaination of more esoteric parts of the source code.

In chapter 4 we will address the issues AGT has both on definition and implementation levels, and discuss the various features that can be improved or added to the language.

Chapter 5 will conclude the thesis and describe future work the author will undertake in order to improve the language.

2. The AGT Programming Language

AGT compilation process consists of 4 main phases:

1. Lexical analysis - tokenization of input source code text
2. Syntactical analysis - grouping of tokens in an abstract syntax tree
3. Semantical analysis - transforming the syntax tree into a context aware semantical syntax tree
4. Type inference - inferring the main function type and all other required types

The process is successful if the type inference step successfully infers the main function type.

We will now describe each phase in great detail, contrasting the newly defined terms against their use in other languages, since the subtlimities sometimes are different.

2.1. Lexical analysis

The AGT Lexical analysis phase is a standard regular grammar parsing process. Lexical analyzer is fed a stream of characters, which get grouped into tokens, according to regular expressions and ambiguity resolution rules.

Lexical analyer tool used for the AGTC was PLY - Python Lex-Yacc (Beazley, a). Python Lex-Yacc is a tool which tries to bring the functionality of well know lexical and syntactical analyzers Lex and Yacc into Python. The main difference is that PLY does not *generate* the concrete lexer and parser. Instead, a PLY user defines the rules fed to PLY by specifying programming constructs like functions that correspond to these rules. PLY then inspects these constructs using reflective capabilities of python, and acts according to these rules when performing lexical or syntactical analysis.

Lets check out some PLY lexer rules used in AGTC.

Listing 2.1: heyy

```
class Lexer():
    states = (
        ('mlc', 'exclusive'), ('strchar', 'exclusive'),
    )
    #...
    tokens = (
        'INTL', 'BOOLL', 'ID',
        'IF', 'ELSE', 'BREAK', 'FOR', 'WHILE', 'RETURN',
        #...
    )
    t_ADD = r'\+'
    t_SUB = r'-'
    t_MUL = r'\*'
    #...
    reserved = {
        'type': 'TYPE',
        'fn': 'FN',
        'let': 'LET',
        #...
    }
    def t_ID(self, t):
        r'[a-zA-Z_][a-zA-Z_0-9]*'
        t.type = self.reserved.get(t.value, 'ID')
        return t

    def t_INTL(self, t):
        r'(\d+)(|i8|i16|i32|i64)'
        return t
    #...
```

Rules for simple operators are given in a string form, in a form of t_<TOKEN> = <REGEX>.

Rules for more complex tokens, like int literals or identifiers are defined using functions. Those same functions can alter the behaviour of the lexer when such a token is parsed. For example, whenever an identifier is parsed, the function `t_ID` looks up whether the identifier is in a list of reserved words, and if it is, changes the tokens type. This neat trick greatly simplyfies the regular expressions (Beazley, b).

PLY documentation states that priority is given to regular expressions defined by functions, in the order they are given, and than to string defined rules, according to their length.

We also employ the *state* feature of PLY, which allows us to change the lexer's behaviour when it encounters, for example, a beginning of a multiline comment. In that particular case, the lexer parses and caches every symbol until it gets to the end of that comment, The state feature is also used for string literals.

An interested reader will find more details about the implementation in the source code supplied with this paper.

This is a great place to introduce reader to special meaning of arithmetic operators. When an expression such as `a + 1` is evaluated, the function call `__add__(a,1)` is evaluated instead. Same goes for all operators, so now we give their equivalent *dunder* functions (the name comes from the Python community).

Arithmetic operators and their correspondant functions methods are:

- + : __add__
- - : __sub__
- * : __mul__
- / : __div__
- % : __mod__

Comparison operators and their correspondant dunder functions are:

- == : __eq__
- != : __ne__
- >! : __gt__
- <! : __lt__
- >= : __ge__
- <= : __le__

The strict less or greater operators include the exclaimation mark in order to prevent some common ambiguities in lexing (which occur due to the presence of angle brackets).

Finally, the last type of builtins are boolean operators.

- & : __and__
- | : __or__
- ~: __not__

Note that, even though we call the last group the *boolean* operators, those operators can be evaluated on operands which are not booleans, as long as the corresponding dunder functions are defined in the program. The same goes for arithmetic and comparison operators; a programmer can use builtins functions for all integer types, but can define additional dunder functions for `rectangle` objects, for example.

Both type of comments, `/*multiline comment*/` and `//singleline comment` are discarded after being parsed.

String and character literals are given in double and single quotes, respectively. They can include escaped characters: (\0, \n, \t, \", \'). Character literal must consist of exactly one character or escaped character.

Identifiers consist of a letter or an underscore, followed by any number of letters, underscores or digits. Reserved words are excluded from identifiers.

Reserved words are:

- ’fn’, ’struct’,
- ’let’, ’type’,
- ’if’, ’else’, ’break’, ’for’, ’while’, ’return’,
- ’true’, ’True’, ’false’,’False’,

Integer literals are characterized by both the value and the size of integer containing this value. Possible integer sizes are 8, 16, 32 and 64.

Example integer literals are:

- `5` (implicitly i32)
- `5i32`
- `516`
- `5i8`
- `5i64`
- `324243` (implicitly i32)

2.2. Syntactical analysis

Syntactical analysis is achieved with a LALR parser implemented in PLY package.

Tokens resulting from lexical analysis are fed into a stack machine which, depending on the state of the top of the stack and next token in the list, either pushes the next token on top of the stack (*shift*) or converts a number of tokens on the top of the stack to another token (*reduce*).

The grammar rule specifications are given in a different way than the one standard for PLY. PLY uses functions by default, while AGTC uses classes which have to adhere to certain structure. These class definitions are turned into functions (using reflective programming) and finally dynamically inserted into syntactical parser class definition. A reader who is interested in details can consult the source code.

The output of a syntactical analysis phase is an abstract syntax tree - a tree structure which denotes applications of grammar rules to tokens. An example syntax tree, which results from the statement `let n = in<i32>();` is given:

Listing 2.2: a

```
Statement(
– statement = InitStatement(
– – name = Expression(
– – – expr = IdExpression(
– – – – id = str(n)
– – – )
– – )
– – expr = Expression(
– – – expr = UnaryExpression(
– – – – expr = ParenthesesCallExpression(
– – – – – expr = Expression(
– – – – – – expr = UnaryExpression(
– – – – – – – expr = AngleCallExpression(
– – – – – – – – expr = Expression(
– – – – – – – – – expr = IdExpression(
– – – – – – – – – – id = str(in)
– – – – – – – – – )
– – – – – – – – )
– – – – – – – – expr_list = [
– – – – – – – – – Expression(
– – – – – – – – – – expr = IdExpression(
– – – – – – – – – – – id = str(i32)
– – – – – – – – – – )
– – – – – – – – – )
– – – – – – – – ]
– – – – – – – )
```

```
– – – – – – )
– – – – – )
– – – – – expr_list = [
– – – – – ]
– – – – )
– – – )
– – )
– )
)
```

This tree consists of a single statement. We won't get into details of the semantics here, since there is a another step which will contextualize the syntax tree and resolve some ambiguities, which couldn't have been resolved with a LALR parser (which PLY uses).

An example of a rule which is used to generate binary expression nodes is:

Listing 2.3: a

```
class BinaryExpression(ParserRule):
    """BinaryExpression : Expression ADD Expression
                        | Expression SUB Expression
                        | Expression MUL Expression
                        | Expression DIV Expression
                        | Expression MOD Expression
                        | Expression LE Expression
                        | Expression GE Expression
                        | Expression LT Expression
                        | Expression GT Expression
                        | Expression EQ Expression
                        | Expression NE Expression
                        | Expression AND Expression
                        | Expression OR Expression
    """

    def __init__(self, r):
        self.left = r[0]
        self.op = r[1]
        self.right = r[2]
```

The shift/reduce conflicts of arithmetic expressions are resolved by specifying a list of operator priorities.

2.3. Semantic analysis

The need for semantical analysis is caused by a fact that AGT's syntax is highly contextual, that is, expressions that appear in one place can have a radically different meaning than same expressions which appear in another place.

For example, in the following code snippet, the expression `a` is used in two distinct contexts.

Listing 2.4: a

```
Statement(
– statement = InitStatement(
– – name = Expression(
– – – expr = IdExpression(
– – – – id = str(n)
– – – )
– – )
– – expr = Expression(
– – – expr = UnaryExpression(
– – – – expr = ParenthesesCallExpression(
– – – – – expr = Expression(
– – – – – – expr = UnaryExpression(
– – – – – – – expr = AngleCallExpression(
– – – – – – – – expr = Expression(
– – – – – – – – – expr = IdExpression(
– – – – – – – – – – id = str(in)
– – – – – – – – – )
– – – – – – – – )
– – – – – – – – expr_list = [
– – – – – – – – – Expression(
– – – – – – – – – – expr = IdExpression(
– – – – – – – – – – – id = str(i32)
– – – – – – – – – – )
– – – – – – – – – )
– – – – – – – – ]
– – – – – – – )
– – – – – – )
```

```
– – – – – )
– – – – – expr_list = [
– – – – – ]
– – – – )
– – – )
– – )
– )
)
```

In the ending of the first line, `a` is a type expression, that is, a compiler is required to produce a type (which the function returns). In the second line, `a` is a value expression, which means compiler is required to produce a value of that expression, rather than a type. We haven't yet specified what means to produce either a type or a value, but those implementation details will be dealt with in the next chapter.

What is important for the understanding of AGT is that

Listing 2.5: a

```
@ dataclass
class InitStatement(FunctionStatement):
    name: str
    expr: ValueExpression

@ dataclass
class MemberDeclarationStatement(StructStatement):
    name: str
    type_expr: TypeExpression
```

2.4. Type inference

Type inference is the most complex phase of AGT compilation process. Before we go heads first into it's description, we will lay down some definitions and motivate them. Type inference also includes LLVM-IR generation.

AGT's type inference engine works as a functional programming language, in a sense that it has to evaluate *type requests*. This evaluation is *pure*; repeated evaluations always yield the same result. In a sense, the type engine is a box which provides two evaluation functions.

2.5. Concrete type request

The first function is a *Concrete type request* (CTR). This function maps a name and a list of concrete types (*type arguments*) into a new concrete type. For example,

1. CTR("char", []) -> CharType
2. CTR("i32", []) -> IntType(32)
3. CTR("rectangle", [i32]) -> RectangleType(with i32 side lengths)

How does the type engine achieve this mapping?

One way is to use `concrete type generators`. They take CTR arguments and decide whether they can provide a type, for example, a bool type generator can check whether the name is equal to "bool" and whether there are no supplied type arguments; if both conditions are satisfied, it can provide the concrete type.

The other way is to iterate over user-supplied struct definitions. For every struct definition whose name matches the supplied name, and whose number of type parameters matches the number of supplied type arguments, the inference is being done. Every sucessful inference is considered in a final decision.

Inference using struct definitions consists of evaluating the body of struct definition, which might need recursive calls to type engine itself. We ensure that these recursions are taken care of; for example, recursive structure definitions will fail.

When both of these methods finish, there has to be **exactly one** candidate type, otherwise, we have an ambiguity, which is an error.

2.6. Function type request

The second inference function is a *Function type request* (FTR). This function maps a name and **two** lists of concrete types (*type arguments* and *value arguments*) into a new concrete type. For example,

1. FTR("main", [], []) -> FunctionType
2. FTR("fibonacci",[], [i32]) -> FunctionType
3. FTR("power", [fast], [i32, i32]) -> FunctionType
4. FTR("power", [slow], [i32, i32]) -> FunctionType

A FunctionType is a rather complex type; it needs to have all information needed for code synthesis, including LLVM code and return type.

By default, every function call is preceeded by copying of its arguments. However, some built-in functions, like copy functions of pointers, should not copy their arguments (because of infinite recursive loops). Only builtin functions can ignore this copying behaviour, and the flag which indicates whether to do so is also a part of a FunctionType.

Let us take a look at the following code snippet, which demonstrates a sort of polymorphic behaviour.

Listing 2.6: a

```
1  struct slow{;}
2  struct fast{;}
3
4  fn power<T>(b, p){
5      type _ = enable_if<T==slow>;
6      ...
7  }
8
9  fn power<T>(b, p){
10     type _ = enable_if<T==fast>;
11     ...
12 }
```

Suppose the type engine is invoked to fulfill a FTR("power", [slow], [i32, i32]). Since "power" is not in builtin functions, TE will resort to using function definitions.

When TE starts to infer a type from the definition at line 4, it will immidiatelly encounter a special construct `enable_if<...>` at line 5. This construct is the main driver of polymorphism in AGT. AGTC will first evaluate it's argument, `T==slow`, which will resolve to `i1` (equivalent of true when used in type context). Enable_if will, since its argument represents truth, act as if didn't exist; it will just assign the type to variable on the left side. Since the variable is a *discard token* (_), the type won't be assigned, but rather discarded.

The user has achieved polymorphic selection using enable_if. He signalled the compiler that this definition of power function fits programmers intentions.

On the other hand, consider the definition of `power` at line 9. In this line, `enable_if` argument will evaluate to `i0` (false in type context). This will cause the TE to abandon this function as a potential candidate for type resolution. Note that this behaviour is not a error (the one which AGTC reports as a mistake), but rather a mechanism for polymorphic selection.

2.7. Lifetime semantics

Lifetime semantics is a broad term which rougly encompasses the following aspects of objects in programming languages:

1. Objects exist in a specific context, which is dependant on their

   lexical environment (the scope)

   runtime (the function invocation)

   For example, parameters of the function exist and can be referenced throughout the function body, but stop existing when that specific invocation of the function terminates.

2. The context in which objects exist is called their *lifetime*

3. Object's lifetime starts with *initialization* (*construction* in some sources).

   Having precise control over construction can allow programmer to restrict usage of the object or set up invariants

4. Object's lifetime ends with *destruction*. Having precise control over destruction allows programmer to deconstruct complex inner workings of the object, without relying on programmer to do it manually. For example, objects can *release* their resources when they stop existing.

5. Transferring or duplicating object's contents to other memory location can also be controlled, via `copy operation`. This allows us to, for example, deep copy the memory array, instead of shallow copying the pointer and thus incidentally creating unwanted memory aliases.

Various other semantics can be implemented on top of lifetime model, we will check out an example called `Ownership semantics` in the last section. Now we will describe how lifetime semantics fit into definition of AGT. Before we get into the details though, we have to throughly define types of values.

Definition 1 (L-value)**.** A value expression is said to a L-value, if the value it evaluates to **can** be referred to from other context.

Definition 2 (R-value)**.** A value expression is said to a R-value, if it is not L-value, that is if the value it evaluates to **can't** be referred to from other context.

For example, every statement `let <v> = ...` creates a new memory location, so the expressions `<v>` following it and refering to same identifier `<v>` are lvalues.

On the other hand, function call expressions such as `gcd(3,2)` are rvalues; their result (function return value) can't be refered to from any other place, apart from this one.

The distinction between L-values and R-values is crucial when reasoning about copy operations. Since R-values can't be refered to from any other context, instead of copying their value, we can `memory copy` their value. The crucial distinction is that `copying` involves calling the `copy` function (which might be very expensive, for example for vectors), and *memory copying* does not copies only the literal memory contents.

Definition 3 (Initializer)**.** An initializer is a function whose name is $_{init,andwhichhasatleastonevalueparameter;apointe}$

Definition 4 (Copy function)**.** An copy function is a function whose name is $_{copy,andwhichhastwovalueparameters;}$

Definition 5 (Destructor)**.** An destructor is a function whose name is $_{dest,andwhichhasonlyonevalueparameter;apointe}$

Check out the following program and it's output:

Listing 2.7: a

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

struct B{
        let member = i32;
}

fn __init__(bp, i){
        type _ = enable_if<bp==@B>;
        type _ = enable_if<i==i32>;

        outnl("Init B started");
        bp!.member = i;
        outnl("Init B finished");
```

```
}

fn __copy__(bp1, bp2){
        type _ = enable_if<bp1==@B>;
        type _ = enable_if<bp2==@B>;

        outnl("Copy B started");
        bp1!.member = bp2!.member;
        outnl("Copy B finished");
}

fn __dest__(bp){
        type _ = enable_if<bp==@B>;

        outnl("Dest B started");
        __dest__(@(bp!.member));
        outnl("Dest B finished");
}

fn main() -> i32{
    let b = object<B>(5);
    outnl(b.member);

        return 0;
}
```

Listing 2.8: a

```
Init B started
Init B finished
5
Dest B started
Dest B finished
```

You can clearly see that we have used `enable_if` construct in lifetime functions too. This is crucial to prevent multiple candidates for different concrete types.

These were the basics of lifetime semantics, we will get to more details in examples section.

3. AGT program examples

This section of the paper will introduce the reader to AGT by presenting, explaining and contemplating about example programs.

3.1. Limited parametric polymorphism

Parametric polymorphism is a possibility of function to be called with different sets of argument types.

Check out the following example:

Listing 3.1: a

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

struct is_int<T> -> R{
    type R = (T==i64 | T==i32 | T==i16 | T==i8);
}

fn fib<T>(n) -> T{
    type _ = enable_if<is_int<T>>;
    type _ = enable_if<n == i32>;

    if (n<=2){
        return cast<T>(1);
    }
    else{
        return fib<T>(n-1)+fib<T>(n-2);
    }
```

```
}

fn main() -> i32{
    let t = 12;
    let f1 = fib<i8>(t);
    let f2 = fib<i32>(t);
    outnl(f1);
    outnl(f2);
    return 0;
}
```

Output:

Listing 3.2: a

```
-112
144
```

3.2. Compile time computation

Check out the following example:

Listing 3.3: a

```
struct fib<T> -> i1{
    type _ = enable_if<T == i1>;
}
struct fib<T> -> i1{
    type _ = enable_if<T == i2>;
}

struct fib<T> -> R{
    type _ = enable_if<T != i1>;
    type _ = enable_if<T != i2>;
    type R = fib<T-i1>+fib<T-i2>;
}

fn main() -> i32{
    let v = type_to_value<fib<i12>, i32>();
    out(v);
    return 0;
```

```
}
```

Output:

Listing 3.4: a

```
144
```

3.3. Ownership semantics

Check out the following example:

Listing 3.5: a

```
fn outnl(a){
     out(a);
     out("\n");
}

struct shared_ptr<T>{
     let item_ptr = @T;
     let count = @i32;
}

fn __init__(sptr_ptr, item_ptr){
     outnl("INIT");
     type T = enable_if_resolve<sptr_ptr!.T>;
     type _ = enable_if<sptr_ptr == @shared_ptr<T>>;
     type _ = enable_if<item_ptr == @T>;

     sptr_ptr!.item_ptr = item_ptr;
     sptr_ptr!.count = heap_object<i32>(1);
}

fn __dest__(sptr_ptr){
     outnl("DEST");
     type T = enable_if_resolve<sptr_ptr!.T>;
     type _ = enable_if<sptr_ptr == @shared_ptr<T>>;

     sptr_ptr!.count! = sptr_ptr!.count! - 1;
     if (sptr_ptr!.count! == 0){
```

```
        outnl("RELEASE");
        heap_free(sptr_ptr!.item_ptr);
        heap_free(sptr_ptr!.count);
    }
}

fn __copy__(sptr_ptr_dest, sptr_ptr_src){
    outnl("COPY");
    type T = enable_if_resolve<sptr_ptr_dest!.T>;
    type _ = enable_if<sptr_ptr_dest == @shared_ptr<T>>;
    type _ = enable_if<sptr_ptr_src == @shared_ptr<T>>;

    sptr_ptr_src!.count! = sptr_ptr_src!.count! + 1;

    sptr_ptr_dest!.item_ptr = sptr_ptr_src!.item_ptr;
    sptr_ptr_dest!.count = sptr_ptr_src!.count;
}

fn make_shared(item_ptr) -> shared_ptr<inner>{
    type inner = enable_if_resolve<item_ptr!>;
    return object<shared_ptr<inner>>(item_ptr);
}

// actual use

fn main() -> i32{
    let a = make_shared(heap_object<i8>(15i8));
    outnl("Before B is initialized");
    outnl(a.count!);
    outnl(a.item_ptr!);
    {
        let b = a;
        outnl("After B is initialized");
        outnl(a.count!);
        outnl(a.item_ptr!);
        outnl(b.count!);
        outnl(b.item_ptr!);
        b.item_ptr! = 13i8;
```

```
            outnl("After B is changed");
            outnl(a.count!);
            outnl(a.item_ptr!);
            outnl(b.count!);
            outnl(b.item_ptr!);
        }
        outnl("After B is destructed");
        outnl(a.count!);
        outnl(a.item_ptr!);

        return 0;
}
```

Output:

Listing 3.6: a

```
INIT
Before B is initialized
1
15
COPY
After B is initialized
2
15
2
15
After B is changed
2
13
2
13
DEST
After B is destructed
1
13
DEST
RELEASE
```

3.4. Memory primitives

Check out the following example:

Listing 3.7: a

```
struct rect<T>{
    let a = T;
    let b = T;
}

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

    type irect = rect<i32>;

    let rect_heap = heap_object<irect>(a,b);

    out(rect_heap!.a);
    out(rect_heap!.b);

    heap_free(rect_heap);
    out(rect_heap!.a);
    out(rect_heap!.b);
}
```

3.5. Memory allocation

Check out the following example:

Listing 3.8: a

```
fn printpos(ptr, pos){
    out(ptr[pos]!);
}

fn setpos(ptr, pos, val){
    ptr[pos]! = val;
}
```

```

fn main() -> i32{
    let A = heap_alloc<i32>(10);
    setpos(A, 3, 123);
    setpos(A, 6, 456);
    printpos(A, 3);
    printpos(A, 6);
    printpos(A, 5);
    heap_free(A);

    // B will sometimes be allocated exactly where A was, and overwrite i
    // of course, this is a memory error, but it will probably continue v

    let B = heap_alloc<i32>(10);
    setpos(B, 3, 0);
    setpos(B, 6, 0);
    printpos(A, 3);
    printpos(A, 6);
    printpos(A, 5);

    return 0;
}
```

3.6. Object oriented example

Check out the following example:

Listing 3.9: a

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

fn len(cptr) -> i32{
    type _ = enable_if<cptr == @char>;
    let cl = 0;
    while (cptr[cl]! !='\0'){
        cl=cl+1;
```

```
    }
    return cl;
}

//

struct string{
    let arr = @char;
    let len = i32;
}

fn __init__(sptr, cptr){
    type _ = enable_if<sptr == @string>;
    type _ = enable_if<cptr == @char>;

    let l = len(cptr);

    sptr!.len = l;
    sptr!.arr = heap_alloc<char>(l);
    while (l>!0){
        sptr!.arr[l-1]! = cptr[l-1]!;
        l=l-1;
    }
}

fn __copy__(sptr1, sptr2){
    type _ = enable_if<sptr1 == @string>;
    type _ = enable_if<sptr2 == @string>;

    sptr1!.len = sptr2!.len;

    let l = sptr1!.len;

    sptr1!.arr = heap_alloc<char>(l);
    while (l>!0){
        sptr1!.arr[l-1]! = sptr2!.arr[l-1]!;
        l=l-1;
    }
```

```
}

fn __dest__(sptr){
    type _ = enable_if<sptr == @string>;

    heap_free(sptr!.arr);
}

//

fn get(sptr, i) -> char{
    type _ = enable_if<sptr == @string>;
    type _ = enable_if<i == i32>;

    return sptr!.arr[i]!;
}

fn set(sptr, i, c){
    type _ = enable_if<sptr == @string>;
    type _ = enable_if<i == i32>;
    type _ = enable_if<c == char>;

    sptr!.arr[i]! = c;
}

fn out(s){
    type _ = enable_if<s == string>;

    for(let i=0; i <! s.len; i=i+1;){
        out(s.arr[i]!);
    }
}

fn main() -> i32{
    let a = object<string>("Hello world!");
    let b = a;

    outnl(a);
```

```
        outnl(b);
        outnl(get(@a,1));
        set(@a, 1, 'E');
        outnl(a);
        outnl(b);

        return 0;
}
```

4. Future improvements

4.1. References

4.2. Syntactic improvements

4.3. Multiple source file support

4.4. Portability

5. Conclusion

Zaključak.

BIBLIOGRAPHY

[Beazley a] BEAZLEY, David: *PLY (Python Lex-Yacc)*. – URL http://www.dabeaz.com/ply/

[Beazley b] BEAZLEY, David: *PLY (Python Lex-Yacc) documentation*. – URL https://ply.readthedocs.io/en/latest/index.html

[Fred L. Drake Jr.] FRED L. DRAKE JR., others: *Python3 documentation*. – URL https://docs.python.org/3/glossary.html#term-duck-typing

[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.