University of Tartu
Faculty of mathematics and computer science
Institute of computer science
Specialty of computer science

Filipp Ivanov

Restricting Python for pedagogical use

Bachelor thesis (6 EAP)

Supervisors: Härmel Nestra, PhD
Margus Niitsoo, PhD

Author:	….....................................	…. mai 2012
Supervisor:	….....................................	…. mai 2012
Supervisor:	….....................................	…. mai 2012

Lubada kaitsmisele

Professor:	….....................................	…. mai 2012

Tartu 2012

Contents

Introduction

This thesis deals with application of the Python programming language to the taks of teaching students the beginning of programming.

Chapter 1 describes the background of the work by stating the shortcomings of using the language to teach the basic computer science courses. It then depicts the proposed solution in general and shows how it relates to the problems. The last section surveys related projects, similar in either goals or means.

Chapter 2 starts with a description of the program and its usage, lising the intended use case for each profile, then proceeds to the main part - high-level overview of the implementation. Section 2.2 defines the data structures used in the filter a describes the role of each of data flow stages. Section 2.3 then describes the evolution of the type system used and algorithms related to type checking stage.

Chapter 3 gives a more detailed overview of the source code, including the reasons behind the choice of implementation technology. Each section corresponds to one of the processing stages in the flow.

Conclusion outlines the possible directions for improvement.

1.Statement of the problem

1.1 Background

The programming language Python [1] is currently used in many universities, including the University of Tartu, as a basis for introductory programming courses. Primary reason for this is the high readability, achieved by using clean syntax and forcing good coding style, the most common example of which is using indentation as a necessary block delimiter instead of a style guideline as in C-family languages. However, there are certain drawbacks in using a modern general-purpose language like Python for educational purposes in comparison to those tailor-made for teaching:

1. **"Syntactic sugar" i.e. convenient additional constructions:** The reason for not sticking to bare minimum of programming constructs is clear as it raises the expressive power of the language and the level of abstraction, allowing already trained programmers to focus on algorithms, not the underlying operations. In order to make use of advanced constructs, however, the students need to understand the basics first. The aim of the first programming courses is not to teach language features, but to learn how to think logically like a programmer.

   For example, Python allows easy checking if integer belongs to a given range:

is_single_digit_positive = 0 < n < 10. Despite being much clearer, it is not as useful for teaching purposes as its expansion which emphasises the logic of comparison (even though the latter is closer to familiar mathematical notation): i**s_single_digit_positive = 0 < n and n < 10.** The same applies to built-in libraries, that contain a bulk of useful methods for list and string objects. For learning purposes, it is crucial that the student will himself learn to implement at least some of the basic algorithms: searching for an element or subsequence, finding maximum, sorting and so on.

2. **Dynamic weak typing.** Shifting type checking to runtime makes quick prototyping and scripting much easier in dynamic languages like Python, but it comes with a couple of disadvantages. Obvious type errors like adding an integer to a string and, more commonly, typing errors are detected only in runtime. They might exist in a rarely-called branch and be discovered only when dealing with corner cases. Losing the result of the run because of a small typing error in a function name or a missing colon can be frustrating for beginners. Static type systems are able to detect most of such simple errors during compile-time.

1.2 Proposed solution

The aim of this thesis is to create a language based on Python which addresses the two issues mentioned above. As we still want to keep the numerous benefits of Python, the new language will be a proper subset of it so every program will still be interpretable by any valid Python distribution. This approach adds some constraints to the semantics, but allows one to implement the new language as a preprocessing filter that performs the necessary checks and then passes the code to an actual Python interpreter. This way only code checking logic needs to be written and, more importantly, existing tools and development environments can be used with the created language. The proposed filter will perform two kinds of checks, dealing with the two problems outlined in Section 1.1:

1. **Forbidding language constructs by constraining the grammar**. Depending on the needed features, the language can be shaped, for example, to allow only a single return statement in a function. It can also be used to disable all for-loops and class definitions.
2. **Statically type-checking the code**. According to the requirement that every program must also be valid in Python proper, we can not add any additional syntax rules, such as type declarations and the checker must rely solely on type inference. Python uses so-called “duck typing” in run-time: if an object has a method foo(), it is usable in any code, which calls foo(), even if its semantics is completely different from intended in code. In other words, suitability of object in expressions depends only on its interface, not the exact type. This, in general, makes it very hard, if not impossible, to typecheck Python code. However, if we

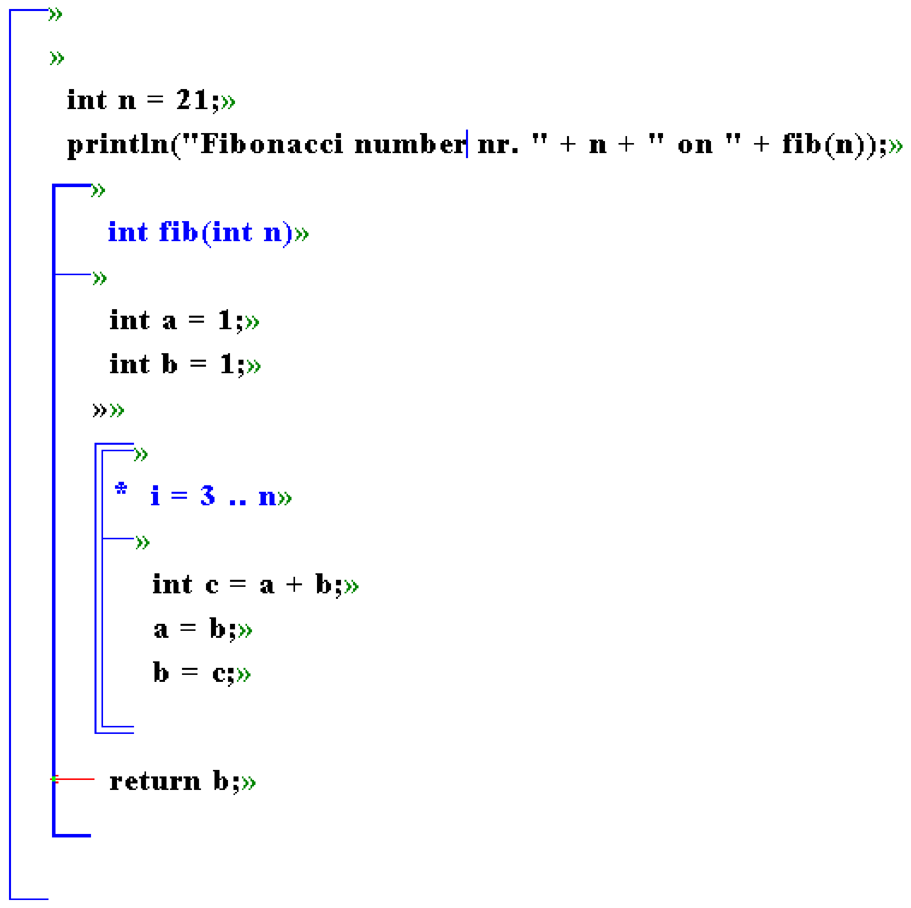
use only a limited subset of constructs (see previous point), a sound type system can be built.

In fact, multiple different *profiles* are defined in the filter, each containing its own rules about grammar, types and additional constraints. As an example, there are no lists or string mutation methods in the core imperative profile.

1.3 Related work

1.3.1 AlgJava

There is a moderately large amount of languages and environments designed for teaching programming. One of them, AlgJava[2], is a result of work done in the University of Tartu by Jüri Kiho. AlgJava is a development package which uses Java as a basis language but adding macroes for simpler input/output and removing complex object-oriented parts that are irrelevant for beginners, helping them focus on the program logic. The primary feature of AlgJava is that the source code is not edited directly as plain text, but in an editor showing the block structure of a program (Figure 1).



```
»
»
 int n = 21;»
 println("Fibonacci number nr. " + n + " on " + fib(n));»
   »
   int fib(int n)»
   »
   int a = 1;»
   int b = 1;»
   »»
     »
     * i = 3 .. n»
     »
       int c = a + b;»
       a = b;»
       b = c;»

   return b;»
```

Figure 1. AlgJava editor showing the source code of Fibonacci number generator

Students do not write control statements directly, but use menus to add templates and fill in the details by hand. Statements which do not open a new block come from Java, with the addition of macros for common operations (e.g. println instead of System.out.println in listing above). AlgJava uses the same method for simplification of teaching as this thesis - forbidding irrelevant elements. However, it is different in the way programming is treated - while AlgJava uses it’s own code format requiring a special environment, we use plain text compatible with every code editor.

1.3.2 Stackless Python

Stackless Python[3] is an extension of Python, which includes primitives for cooperative multithreading in form of efficient user-level microthreads. All language features of Stackless are available as library elements in code. However, the difference between the common kernel-level multithreading and microthreads is high enough that Stackless requires its own implementation of interpreter. Despite having the same grammar as Python, Stackless programs differ greatly from their canonical counterparts, embracing concurrency as almost primitive lightweight element of the language, as all scheduling is performed in userspace.

Although the aims of Stackless are orthogonal to this project, both use Python as an underlying language, although complement it in different ways: this project builds a preprocessor whereas Stackless has a modified runtime.

2.Description of the filter

2.1 Overview of functionality

The program is intended to be used in teaching basic programming and the main use case is the homeworks. The teacher not only assigns the exercise, but also which language elements are available, dependent on the current lecture topic. Many tasks can be done easily using Python's wide range of features, but this way the teacher can be sure that students apply the knowledge used in class and not just use generic pre-programmed library methods. On the student side, the filter checks that the solution is relevant to the topic and prevents confusing type errors, showing what the error is and where it did occurs before run-time execution.

The filter is transparent so the usage is almost the same as with the normal, complete interpreter. The only thing needed is to specify the profile (sublanguage used) using --profile=<name> argument

```
bpython --profile=ImperativeExtended test.py
```

will check that test.py conforms to the Imperative Extended profile rules and if so, pass the code to the Python interpreter together will all parameters excluding --profile.

There are two different lines of profiles: those containing imperative elements (state modifications, loop structures) and functional ones. Both have instances of increasing complexity. Those are not disjoint, as the language is useless without core features, which will be common to both directions. Even though the functional profile is outside the scope of this work and is not subject to implementation, it is discussed in the text to put language feature into context and give a basisy for future development of the system.

Basic *ImperativeCore* profile specifies mostly used elements of the language. It is already Turing-complete and suitable for teaching - its capabilities are those usually used in pseudocode in textbooks:

- Literals - integers, floating point numbers, strings and Boolean values
- Arithmetic operations on numbers (+, -, *, /, //, %) and string concatenations
- Logical primitives (and, or, not) and comparison operators (<, <=, ==, =>, >)
- Input/output functions: **print(), input()**
- Type conversion (may be confusing for students, but needed because of the type of **input()**): **int(), float(), str()**
- String methods without side effects: **strip(), replace(), find(), count(), endswith(), lower(), upper(), capitalize()**

- String type predicates: **isalpha(), isalnum(), isdecimal(), isdigit(), islower(), isnumeric(), isspace(), isupper()**
- Conditional statements **if .. elif .. else**.
- While-loop
- Variables and assignments - as there is only global scope, variable holds its type throughout the whole code

Imperative core is sufficient enough to express most numerical algorithms which usually serve as examples in the early programming courses: primality checks, solving equations etc.

When the students are introduced to the array structure, the level of language needs to be raised to an *ImperativeExtended* profile. In addition to previous elements, it also supports lists, their direct indexing, and basic Python slice syntax where **arr[k:l]** extracts sublist, which contains elements from index k inclusive to index l exclusive. Algorithms on this course stage (sorting, searching etc) require additional control structures such as **break** and **continue**. Commonly used primitives are added: functions **len(), sum(), range(), list()**, list mutation methods **append()** and **reverse()#** and string methods **replace()** and **split()**.

ImperativeAdvanced is a slight extension of *ImperativeExtended*, which adds only the for-loop. At this stage in learning there is not much point in forcing students to further traverse lists on low level by manually increasing and controlling indices.

2.2 Overview of the implementation

The working principles of the filter resemble those of the front-end of most language translators. It takes as input the program code (along with control switches) and passes it through several stages of analysis (Figure 2)

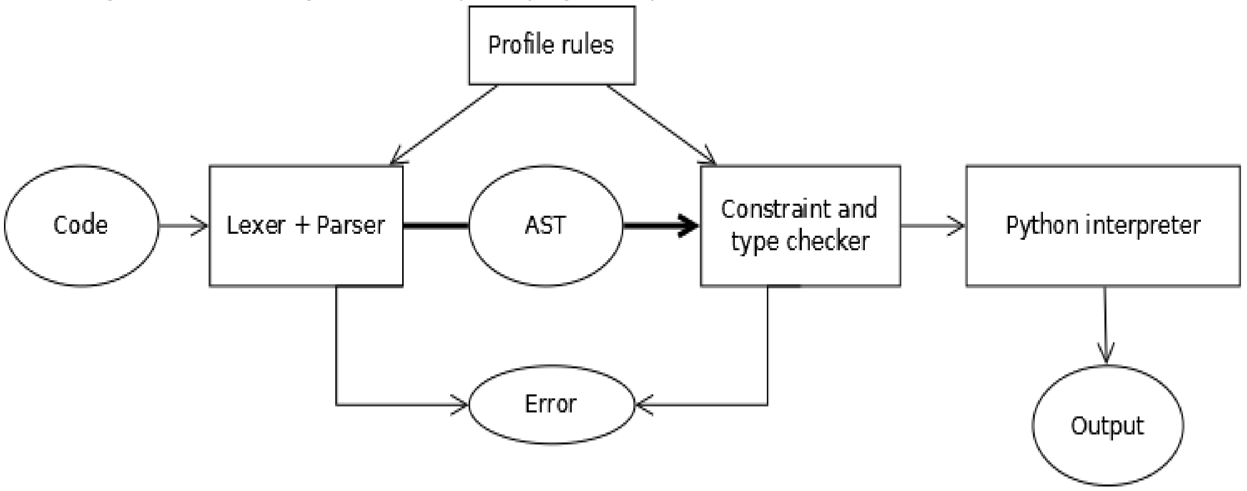


Figure 2. Processing stages of the filter

Lexical analyzer (lexer / scanner). As the program code is externally just a plain text file, its internal structure needs to be inferred. The task of the lexer is to separate the text into basic lexical elements of the language, so-called **tokens**, such as literals (strings, numbers and other constants), identifiers (names for variables, functions, classes), keywords, delimiters and so on. Additionally, the position of tokens in the source file is attached, to make error messages more readable. At this stage semantically irrelevant data is removed, including comments and formatting whitespace. Lexer can only detect the most trivial errors, such as unmatched quotes and broken indentation.

The lexical structure of Python programs[4] is not that complex, but contains a couple of difficult nuances. One of them is the availability of different types of string tokens, each with its own delimiters, leading to their grammar being not regular, but context-free, meaning that the usual way to deal with scanning - regular expressions - are not suitable for all literals. Floating point numbers also have some nontrivial forms. The other difficulty is the reliance of language on indentation in the source file to define blocks. However, the Python specification contains clear rules of inferring block scopes from indentation changes, so this was not a problem.

Lexer takes the source code and outputs the stream of tokens to the next stage. For example, this code:

```
def foo(x):
    return x == 0

print(foo(2)) # output
```

gets tokenized to the following stream:

```
[(Keyword "def",(1,1)), (ID "foo",(1,1)), (Delimiter "(",(1,1)), (ID "x",(1,1)), (Delimiter ")",
(1,1)), (Delimiter ":",(1,1)), (NewLine,(1,1)), (Indent,(2,2)), (Keyword "return",(2,2)),
(ID "x",(2,2)), (Operator "==",(2,2)), (IntegerLiteral "0",(2,2)), (NewLine,(2,2)),
(Dedent,(4,4)), (ID "print",(4,4)), (Delimiter "(",(4,4)), (ID "foo",(4,4)), (Delimiter "(",
(4,4)),(IntegerLiteral "2",(4,4)), (Delimiter ")",(4,4)), (Delimiter ")",(4,4)), (NewLine,
(4,4))]
```

Note that both the cosmetic empty third line and the comment did not influence the stream (there is only one NewLine token between function definition and print() call). The number pairs denote the line range where the token was detected. Almost every token in lexer is located on a single line, but in the next stages those ranges are combined to precisely specify where the error occurred. Literals contain their token string, which is actually unneeded in the filter (only the type is needed), but can be useful in the future.

Syntactic analyzer (parser). The program code is not linear, but highly structured. The goal in this stage is to infer this structure in form of **abstract syntax tree (AST)**. The AST shows the role of each language element and the sub elements it is constructed from. This tree is the basis for any program analysis, as it is basically a decoded program text, available for the machine processing. As an example, the parsing of the code above results in the AST on Figure 3:

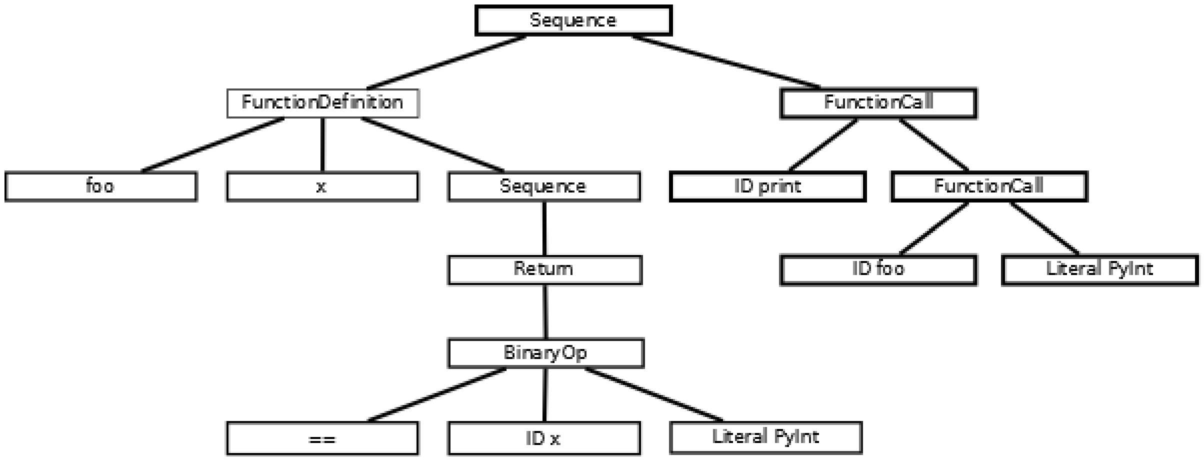


Figure 3. Abstract syntax tree example

The parsing of token stream is done using top-down approach, where the tree is built starting from root node. The parser tries to choose grammar productions that match the leftmost unparsed token in the stream. The complete Python grammar[5] is quite big, but only parts allowed in profiles are relevant, which helps to reduce the complexity. With a couple of exceptions, only one rule can be chosen, making parser mostly predictive, even though backtracking is still needed in some cases.

Constraint checker. Each profile contains a function that enforces grammar constraints, forbidding constructs as stated in the Section 1.1. It walks the syntax tree, checking whether each node is valid with respect to the profile rules. For example, unified grammar defines the simple statement as:

```
simple_stmt ::= assignment_stmt | return_stmt | break_stmt | continue_stmt |
expr_stmt
```

However break keyword is forbidden in core profiles. Code using it will still pass the parsing stage, but resulting AST will not satisfy constraint function. This check is not limited to grammar contraction, but can do additional analysis on nodes, e.g. make sure that each branch in the procedure has a return-statement.

Type checker. The main analysis performed by the filter is typechecking, which tries to assign a type to each expression and checks whether those are consistent. Type checker can be the same for each profile, as the type system changes only slightly between sublanguages. For example, even though the function definition is not available in the most basic profile, type checker can contain function-related code - filtering definitions out will occur on parsing stage. As type checking is the main feature of the system, we will describe it in more detail in the following section.

2.3 Type checking

Languages can be typed either explicitly or implicitly. In the explicit systems, element types are assigned by the programmer in form of variable, function and other declarations. Examples include most of currently used general-purpose compiled languages, such as the ones belonging to C syntax family - C, C++ and Java. The compiler then checks if the usage conforms to declarations - if a string variable is passed into function requiring an integer, a type error is raised. However, manually assigning a type can be cumbersome and in some cases can harm readability: for example, prior to C++11[6], programmers were required to name full iterator types, even though those were already completely determined by template arguments of collection classes:

```
list<string> collection;
...
list<string>::iterator i = collection.begin();
```

In big systems with heavy use of templates declarations could become quite long. The same problem haunts all sufficiently powerful explicit type systems. The logical step further is to let compiler infer at least some, if not all, types from the code. That is the basic idea of implicitly typed languages (though there is another case of dynamically typed code, where types are checked in runtime without neither annotations nor inference required, but this thesis is concerned with static systems for the reasons outlined in the first part). The C++11 standard added a basic inference means to the language, letting the programmer to write the code above as:

```
list<string> collection;
…
auto i = collection.begin();
```

This does not make C++ implicitly typed, but helps to reduce the visual noise and demand on programmer. Some languages provide almost full support for type inference: not only do variables not require declarations, but also the types of functions' formal arguments are determined by the compiler. For example, Haskell can infer the type of function

```
fac x = if x == 0 then 1 else x * fac (x -1)
```

as (Num a) => a → a, meaning "function, which takes a numerical value and returns the value of the same type". Even though the actual factorial function should only be applied to integral arguments, this constraint is nowhere to be found in code, so the inference engine produced the **most general type** for the expression.

Inference and checking are not two disjoint processes - inference algorithms depend on type checking, using its information to guide type assignments.

The described filter implements complete type inference for restricted subsets of Python. Next sections will describe difficulties and solutions for type systems of increasing complexity, leading to the final algorithm used in the filter.

2.3.1 Simple typing and basic polymorphism

The easiest profile to check is the imperative core. Distinctive feature of it is the lack of means to define functions. Programmer is left with a set of basic language elements (literals, operator, while-loop and conditional statement) and few predefined functions related to input/output. Each variable gets its type from the first assignment to it. The assigned value can only be:

1. Literal - they carry their exact type as detected by lexer
2. Expression - all functions and operators are predefined and the result types of their applications can be uniquely determined from the types of the arguments.
3. Another variable - should already carry its type, otherwise it is a checking error

So the type of each variable can be ultimately derived from literals and applications. Type system at this point is simple:

1. Primitive types (all except **PyVoid** have the same semantics as respective Python dynamic ones):
    a. **PyInt** - Integers
    b. **PyFloat** - Floating point numbers
    c. **PyString** - Strings
    d. **PyBoolean** - Boolean values
    e. **PyVoid** - Type without values, used as the type of statements (e.g. while-loop)
2. Compound types - only functions of the form **Type*** → **Type** - list to the left of arrow describes the types of function arguments, the right-hand annotation is the return type of the funcion.

Operator application rules, such as (**PyInt** + **PyFloat** → **PyFloat**) can be hardcoded. In the setting of simple typing the checking algorithm is simple, defined by induction on the AST node type (Figure 4). The only state maintained is the symbol table - dictionary mapping variable names to their types.

---

1. Literal - return the literal type
2. Variable **x** - check whether symbol table contains the entry for **x**. If it does, then return associated type. Otherwise, raise error, because the **x** used before anything was assigned to it.

3. Assignment of expression **e** to variable **x** - compute the type **t** of **e**. If x is not already in symbol table, add entry **x** = **t** to it. Otherwise, compare **t** to the known type of **x**. If those two are equal, return **t**, if not, raise error about type mismatch, as **x** has different types in different code sections. As only function and class definitions open a new naming scope in Python, any variable must have a single type throughout the whole code in the imperative core.
4. Function or operator application - check if the arguments have valid types and compute the resulting type. In function case this is straightforward, as there is only one possible type for each parameter and result type is defined. Operators require some sort of table enumerating all possible argument type combinations.
5. If statement - check all branches, return **PyVoid**.
6. While-loop - check that condition has type **PyBoolean**, check loop body and return **PyVoid**

Figure 4. Type checking algorithm in imperative core profile

However, even in this profile this is not sufficient. For example, Python output function **print()** takes a value of any type as the argument. Such functions are called **polymorphic**. In this simple case, the workaround used is to define additional fictive type **PyAny**, which matches each of other types. So the type of **print()** is **[PyAny] → PyVoid**.

2.3.2 Hindley-Milner type system

Even with the addition of **PyAny** the type system is not expressive enough. It now allows for correct usage of functions, but is also too permissive. Consider identity function **id :: [PyAny] → PyAny**. It can be applied to any value, but return type is not specified, it will always be **PyAny**, when it should be **PyInt** for **id(5)** and **PyBoolean** for **id(True)**. This error even in the best case propagates further: for example, type of

```
5 + id(5)
```

will depend on implementation - as **PyFloat** matches **PyAny**, the result of **id(5)** can be matched to **PyFloat**, adding of [**PyInt, PyFloat**] yields type **PyFloat** for the whole expression, when the actual type should be **PyInt**. When we get to the actual inference, errors become more critical.

The problem can be stated as the relation between types in function signature. The return type of the **id()** is the same as the argument type, but the type system does not allow constraining types this way. The solution for this is to introduce **type variables** into signatures, which perform the same role as variables in formal logic: they can be substituted for concrete types. The signature for **id()** can now be correctly expressed as **forall a : [a] → a**, where **forall** is **universal quantifier** : **forall a : t** means that **a** ranges over all possible types. It is now clear for type checker that argument and result types for **id()** must be the same. When the function is applied, its signature is matched against arguments, **instantiating** the function type. For example, some instances of **id()** type are:

id :: [PyInt] → PyInt
id :: [PyFloat] → PyFloat
id :: forall a : [([a] → a)] → ([a] → a)
id :: forall a : forall b : [([a] → b)] → ([a] → b)

As seen in the example, type variables can be substituted not only with primitive types, but with compound types as well. Last type shows that multiple variables are possible, but we want to avoid signatures similar to this:

forall a : [forall b : ([a] → b)] → (forall b : ([a] → b)

Type systems that allows such arbitrary quantification are called System F[9]. However, the fully implicit type checking for it is undecidable, and such power is not needed for our goals. Restricting **forall** to top-level, we get type system known as **Hindley-Milner Type System(HMTS)**[7]. HMTS is perfectly suited for typing Python, as it does does not require explicit type annotations and has a linear type inference algorithm. As it is defined for lambda-calculus, some minor adaptations for this case are needed, but they preserve all the good properties of HMTS.

Inference in HMTS is not as straightforward as in simple typing[8]. Firstly, additional terms must be defined. As in formal logic, there is a notion of **free** variables, which are those type variables in signature which are not bound by some quantifier. This is not possible in top-level, but happens in inner expressions: **forall a : [a, Int] → a** has no free variables, but **[a, Int] → a** contains free type variable **a**. Non-free (bound) variables can be safely renamed in signature, if the new name is not used there (this can be refined by forbidding only those names, which are free in signature, but it is easier just to generate a new unique name).

Next, the HMTS operates in terms of type **substitutions** and **renamings**. Renaming of type **forall a : t** generates new unique type variable **b**, replaces all free occurrences of **a** in **t** with **b**, getting type **t'** and yields **forall b : t'.** Substitution is a mapping from type variables to types. After applying substitution **{a → e}** to type **t** we get a new type instance **t'** of **t**, where every occurrence of **a** is replaced with **e** (written as **t' = t[a\e]**). This operation has many subtle nuances and needs to be defined formally by induction on type **t** in Figure 5.

1. Primitive type - return **t**
2. Type variable **b** - if **a = b**, then return **e**, else **b**

3. Function type **[a1,a2,..,an] → b** - apply inductively to arguments and return function of the results
4. Generic type **forall b : u** - rename to **forall c : u'**, apply substitution to **u'**, getting **u''** and return **forall c : u''**

Figure 5. Substitution algorithm

Another crucial algorithm is Robinson's unification, which (in this application) takes two types and attempts to find a substitution (**mgu - most general unifier**), which will make them match. For example, mgu of **[a] → PyInt** and **[PyBoolean] → c** is **{c → PyInt, a → PyBoolean}**, because when applied to both terms it will yield **[PyBoolean] → PyInt**. As before, unifications is defined by structural induction of argument types in Figure 6.

1. If both are variables, **x** and **y** - if **x = y**, then return empty substitution, else **{x → y}**
2. If one is variable **x** and other is a function **f**, then check whether **x** is a free variable in **f**. It so, then raise error (occurs check excludes infinite types). Otherwise return **{x → f}**
3. If one is variable **x** and another - primitive type **t**, return **{x → t}**
4. If both are primitive types - return empty substitution if equal, raise error otherwise
5. If both are functions - check that argument counts are the same. If they differ, raise error. Otherwise, try to unify arguments and result type, then unify resulting substitutions

Figure 6. Unification algorithm

Now, the HM itself takes a type context and an expression to type as an input. **Type context** is a set of assumptions about types of identifiers (both variables and functions), the same as the symbol table in the simple typing. The algorithm returns not only the type of the statement, but also the substitution, which needs to be applied to context to get valid set of assumptions. For example.

infer(**{foo : a}**, foo(5)) → (**c**, {**a** → ([**PyInt**] → **c**})

The unification algorithm detected that **foo** is a function, so **foo :: [b] → c**, where **b** and **c** are new unique variables and changed the context to **{foo : [b] → c}**. Then it unified argument type of the function with the type of the passed value, getting most general unifier **{b → PyInt}**, which then was applied to the current context **{foo : [b] → c}**, getting final context **{foo : [PyInt] → c}**. Types of **foo**: **a** and **[PyInt] → c** were unified to get substitution **{a → ([PyInt] → c)}** and the type of the whole application expression **c**.

The HM (Figure 7) uses structural induction on the terms to get type information about expression **e** in context **ctx**.

1. If **e** is a variable **x**, then check whether there is an assumption **x : t** in **ctx**. If no, then this is a free identifier, which corresponds to the uninitialized variable in the code, therefore error is raised. If **t** exists, then rename **t** to **t'** and return (**{}**, **t**)
2. If **e** is a constant with type **t**, return (**{}**, **t**)
3. If **e** is the definition of function named **f** taking arguments **x1,...,xn**, then generate **n+1** new type variables **t1,...,tn,ret_t** and recursively typecheck the body of the function in the context **ctx + {f : tf, x1 : t1,...,xn : tn}** to get substitution **s1**. The type of the body does not interest us, as it will always equal **PyVoid** - body is a sequence of statements and does not produce a value. Instead all return statements are treated as assignments to a fictive variable with type **ret_t** and types substituted for **ret_t** are unified, getting mgu **s2**. Those substitutions are combined, getting **s = s2 . s1** and the pair **(s, [s(t1),...,s(tn)] → s(ret_t))** is returned, with the function type generalized (by binding all its free variables with universal quantifiers, e.g. **[a,PyInt]** → **a** becomes **Forall a : [a,PyInt]** → **a**).
4. If **e** is the application of function **f** to arguments **x1,...,xn**, first infer type of **f**, getting **(sf, tf)**. Then infer argument types in context **sf(ctx)** to get **(s1, t1),...,(sn, tn)**. Compose **s1,..,sn** to get substitution **s**. Then create new type variable **ret_t** and unify **s(tf)** with function type **[s(t1),...,s(tn)]** → **ret_t**, getting substitution **v**. Return **(v . s . sf, v(ret_t))**. The substitution **s** here is a unifier matching function argument to their actual types. We can not use **f** type directly, as it can not just an identifier, but also an expression return function.

Figure 7. Hindley-milner type inference algorithm

HM is not directly used in the filter, but is a basis for its future extension with functional elements (starting with procedure definitions). The rules will need to be further adapted to allow assignments and function definitions, as the canonical algorithm operates on lambda-expressions. However, as shown in [14] many familiar constructs can be easily translated into lambda-calculus.

2.3.3 Constrained types

Hindley-Milner is expressive enough to type any valid expression in the language. However, it is not sound: some code containing logical type errors can still pass checks. Consider following function:

```
def max(x, y):
        if x > y:
                return x
        else:
                return y
```

The HM will infer the type **forall a : [a,a] → a**. However, then those terms are accepted by type system:

```
max('foo', 'bar')
max(max, max)
```

even though they have no meaning. The problem is that HM is too permissive: variables range over all types, only constraints are between different types in the signature. What is needed is the capability to place additional restrictions on type variables, so that they would instantiate only to a subset of all types. In general case, this would require types of form **forany a in [t1,..,tn] : t**, meaning that **a** can only be one of **t1,...,tn**.

However, as the type system in the filter is implicit, the problem can also be solved by defining a set of constraints and declaring which primitive types hold any of them, as in Table 1.

Constraint	**Description**	**Instances**
Num	Numerical types supporting addition, subtraction, multiplication, division and comparison.	**PyInt, PyFloat**
Seq	Sequences: support len() and indexing	**PyString, PyList t**

Table 1. Example of simplified bounded quantification

2.3.4 Final type system

To summarize, the type system for Python consists of following:

1. Primitives: **PyInt, PyFloat, PyString, PyBool, PyVoid, PyVar(name)**
2. Fictive types: **PyAny, PyAnyOf Type***
3. Functions: **Type* → Type**
4. Homogeneous lists: **ListOf Type**
5. Ranges: **PyRange**[1]
6. Universally quantified types: **forall x : Type**
7. Boundedly quantified types: **forany y in Type* : Type**

[1] Ranges are included as the return type of function **range()**, because since Python3 it stopped returning lists, switching to the object of special class **range**

3. Filter Implementation

3.1 Choice of Programming Language

Haskell[10] is a functional programming language with a kit of distinct features:

- **Referential transparency (purity)**: there is no state. In place of traditional variables Haskell has bindings, which are more related to the mathematical concept of the variable - synonym for a value. Once the value is assigned to the binding, it can not be changed. Pure functions compute their output based solely on input taken, without referencing any global state. At any time in the execution, pure function will return the same output if the same input is provided. This discards a lot of problems, including, for example, traditional complexities of locking in concurrency. Still, many times non-pure code is needed, most commonly related to input/output (keyboard input function can not be made pure by definition - its result depends on the user's actions). Haskell provides means to write it and maintains a clear separation between pure and impure code.
- **Very expressive strong type system**: as the language has long been a testing ground for type-theoretic research, its type system is more sophisticated than in most mainstream languages, allowing, for example, higher-order types (similar to C++-type templates), variant types (tagged unions) and type classes (one of the possibilities to incorporate bounded quantification). Additionally, there are many extensions to the type system[11], which can be activated to further increase the power of the system.
- **Lazy evaluation**: expressions in Haskell are evaluated only when their value is actually needed. This allows to use some programming practices too costly or even impossible in strictly evaluated languages, e.g. infinite structures. However, lazy semantics has some pitfalls: impure code may depend on the order of execution, which is implementation-dependent, reasoning about resource usage is complicated in lazy programs and so on. Thus Haskell also provides primitives to enforce evaluation.
- **Clear syntax**: constructs in Haskell are designed in the way that reduces the size of boilerplate code. For example, as the most common operation is Haskell is function application, it does not require brackets or commas: f x y is an application of function f to value x and y. Similarly, type definitions are concise. This encourages programmers to think in functional style and create higher-level abstract structures without hesitation.
- **Big standard library**: Haskell distributions contain a lot of packages. There are task-specific modules, like in Python or Ruby (e.g. JSON serialization, command-

line parsing, concurrency), but Haskell also has a lot of packages supporing abstractions. For example, the Control.Applicative module provides functions to operate on types with specific properties (applicative functors), without actually doing anything practical. There is also a packet database - Hackage, similar in functionality to repositories of Unix-family operating systems.

Haskell was chosen as the implementation language for several reasons:

1. The theory behind static analysis lends itself nicely to functional paradigm. As described in section 2.2, the filter is essentially a series of processing stages: lexer, parser and type checker, which serve as black boxes, calculating output based solely on the input taken. This kind of data-flow algorithms can naturally be modelled by the composition of pure (side-effect-less) functions representing different stages in computation.
2. The analysis often involves structural induction (for example, see algorithms in Section 2.3) and this kind of conditioning can be expressed in Haskell with minimal amount of excessive code by pattern matching mechanisms. The code written in this way is closer to the original declarative algorithm definitions that, for exmple, the Java code with similar semantics would be. This makes it easier to maintain. To achieve similar properties in Java, additional methods are required, such as the Visitor design pattern[13], and it even those do not allow for as clean a code as Haskell does.
3. Overall, Haskell provides a great range of means to increase abstraction and modularity, e.g. higher-level functions and type classes. This huge expressive power helps to cut down the code size, which leads to easier maintainability.
4. The author has been familiar with the language on an intermediate level for a long time and wanted to try using Haskell (and functional programming in general) outside of toy projects.

3.2 Lexer

The whole lexer resides in the source file Lexer.hs. It defines the key Token type, exporting it and the main function lexer, which takes a list of source code strings and returns a corresponding token stream.

Function processLines removes comments and adds numbers to the lines, which will later be used to show locations in error messages. Then empty lines are removed and consecutive lines are joined when needed (in multiline expression and lines ending with backslashes). Its output is the list of meaningful enumerated strings. It is further processed by the function addIndentation, which expands tabs and adds indentation levels to each line.

After that, each line is tokenized by the function tokenize. It uses the list tokenMap which consists of pairs of functions: detectors and extractors. Detector is a predicate on string, which holds only if some prefix of the string is a valid token. Extractor extracts this token, returning it and the rest of input string. For example, after the extractor for a number token would return (IntegerLiteral “1”, “+ 2”), when run on the string “1+ 2”. Using this list tokenize extracts all tokens from the code line. All this code is an example of Haskell’s expressive capabilities mentioned above: tokenize code is just one line long, using higher-order list generating function unfoldr for the standard library.

Next stage is specific to Python - function inferIndentationTokens processes indentation data received from previous steps to get block starting and ending tokens (Indent and Dedent). It also finally concatenates lines to get a uniform token stream. The last phase unifies adjacent string literals accordingly to the Python lexing rules.

3.3 Parser

File AST.hs declares the type of Abstract Syntax Trees (AST) and defines a function extractNodes, which returns the list of the nodes in AST which satisfy given predicate. This generic function is primarily used to enforce syntactic constraints of the active profile. Actually AST type is used no directly, but in conjunction with an integer pair, which describes, on which lines the construct corresponding to the tree node resides in the original source file.

The parser is in file GenericParser.hs. Parser is generic, because it does not use profile data, but analyzes code according to the union of profiles’ grammars: forbidden constructs are detected later, operating on the ready AST, not while parsing.

Parser is programmed using Parsec[12] library, which implements monadic parsing techniques, containing basic general parsers and ways to combine them into more specific ones. Parsec can be loosely called an embedded domain-specific language inside Haskell programs, as it allows defining parsers declaratively, with the executable specification in almost one-to-one correspondence with documented grammar rules: parser for

```
simple_statement ::= assignment_stmt | return_stmt | expression_stmt
assignment_stmt ::= id “=” expr
```

is written in code as

```
simple_stmt =  try assignment_stmt <|> return_stmt <|> expr_stmt

assignment_stmt = do
     id <- identifier
     delimiter "="
     rvalue <- expr
     returnASTLoc (AST.Assignment id rvalue) id rvalue
```

Grammar is almost directly translated to code, with some details, such as try-combinator to support backtracking (as expression and assignment can both start with identifier) and last line which constructs AST node and combines line ranges for subexpressions.

3.4 Profiles

The Profile type is defined in the file Profile.hs. Each of the profiles consists of the function, which takes an AST node and returns True, if the represented construction is forbidden in this sublanguage, and another function, which takes an AST and checks its types.

The file ProfileCommon.hs declares a data type for errors used in the type checkers. CheckErrorData holds information about the nature of the error: for example (IfNonBooleanCondtion t) means that the condition in some if-clause has a type different from **PyBool**. CheckError holds this data and can additionally contain location in code where the error was detected. There are also some helper functions related to error handling (for example, converting errors to human-readable messages) and collections of language primitives: groups of functions and methods, such as pure string methods, or input\output functions) in the file.

PyType module (PyType.hs) declares all the Python types in the system, functions to transform them into strings and equality relations between them.

The actual profile-related code is in the source file: ImperativeProfiles.hs, which uses algorithms described in section 2.3. The implementations require ways to throw and handle errors, and some relevant state needs to be threaded throughout the checking function. Directly implementing those features in functional setting would lead to the large amount of maintenance code unrelated to the logic itself. For example, there is no state in pure functional programming, so it would be necessary to pass and return in in any function. However, this problem is long solved in Haskell community by using abstract types called **monads**. Monad represents the nature of the computation - State monad allows to implicitly thread state while the Error monad adds exception support. Haskell supports monadic code on the syntax level, so all the plumbing is hidden - algorithms are still directly translated to the code.

3.5 Program itself

Entry point for the filter is the function main in the file Main.hs. It parses command line arguments and analyses the code in the chosen profile.

Conclusion

As the result of the thesis, several sublanguages of Python suitable for teaching basic programming skills were defined. The filter system created ensures that the code conforms to the grammar of the sublanguage and checks types. Type checker uses Hindley-Milner type system with bounded types.

There is a number of directions to develop the system further. The most promising of them would be implementing a functional profile (by adapting the Hindley-Milner algorithm to the Python). Currently method calls are translated on parsing stage to functions with an explicitly passed self object. This lack of distinction between functions and methods can lead to some invalid expressions passing the type checker. Solution to this could be a part of a typing algorithms for an additional profile, which enables object-oriented primitives, such as class definitions. This is not the only way to extend the type system, though - for example, distinguishing between pure and effectful functions could allow to statically analyse the code better.

Profiles do not support module imports. Adding this capability would require the filter to infer the module interface from its source code. As doing this at runtime is too costly, the feature would require some additional infrastructure.

Error messages are currently built in, so they can not be translated without rebuilding the program. One possibility of improving the solution would to externalize them and then allowing a choice of language as a command line argument. The same could be done in a limited way to profile definitions by moving them to configuration files or even by creating a domain-specific language to specify grammar constraints.

4. References

[1] Python programming language
http://www.python.org/
[2] Amadeus AlgJava
http://www.cs.ut.ee/~kiho/AlgJavaHome.html
[3] Stackless Python
http://www.python.org/dev/peps/pep-0219/
[4] The Python language reference - 2.Lexical analysis
http://docs.python.org/py3k/reference/lexical_analysis.html
[5] The Python language reference - 9.Full grammar specification
http://docs.python.org/py3k/reference/grammar.html
[6] Working draft, standard for programming language C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
[7] Luis Damas, Robin Milner - Principal type-schemes for functional programs
Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, pp. 207–212
[8] Ian Grant - The Hindley-Milner type inference algorithm
http://ian-grant.net/hm/hindley-milner.pdf
[9] Benjamin C. Pierce - Types and programming languages, second edition
ISBN 0-262-16209-1
[10] The Haskell programming language
http://www.haskell.org/haskellwiki/Haskell
[11] The Glorious Glasgow Haskell Compilation System User's Guide, version 7.4.1, chapter 7: GHC language features http://www.haskell.org/ghc/docs/7.4-latest/html/users_guide/ghc-language-features.html
[12] Parsec: Direct style monadic parser combinators for the real world
Daan Leijen, Erik Meijer - Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht, 2001
[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Design patterns: elements of reusable object-oriented software
ISBN 0-201-63361-2
[14]Guy Lewis Steele, Jr., Gerald Jay Sussman - Lambda: The Ultimate Imperative
MIT AI Lab. AI Lab Memo AIM-353. March 1976

Web pages and linked documents were last checked at 13.05.2012

5. Addendum

The source code for the system is provided on the DVD

Pythoni kitsendamine õpetamiseks

Bakalaureusetöö
Filipp Ivanov
Resümee

Programmeerimiskeel Python on laialt kasutatud esimese keelena informaatikaaluste õppimiseks. Selleks on hulk põhjusi, mille seas on vajalikud loetavus ja arusaadavus. Kahjuks on Pythonis kui üldotstarbilises keeles omadusi, mis rikkuvad tema sobivust selle ülesande täitmiseks. Käesoleva töö tulemusena on loodud filter, mis töötab Pythoni interpretaatori peale, et nende mõju leevendada.

Esimene probleem pärineb Pythoni süntaksi mitmekesisusest: struktuurid on mugavad kasutamiseks ja peitvad allolevaid põhiprintsiipe. Paljude ülesannete jaoks, mille uus programmeerija peaks ise suutma teha, leiduvad keeles kättesaadavad primitiivid. Tudengil võib tekkida mulje, et programmerimine on peamiselt keele elementide rakendamine. Aluskursuste eesmärk on aga õpetada mitte konkreetset keelt, vaid loogilist mõtlemisviisi. Seega liigne süntaks ainult häirib õppimist.

Loodud süsteem lubab kitsendada keele võimaluste hulka. Enne interpretaatorile lähtekoodi andmist kontrollib ta, kas mingid rakendatud keele elemendid on keelatud kasutamiseks, ja kui see on nii, näitab, miles on viga. Seega õpetaja saab anda sooritamiseks ülesandeid, mille fookuseks on mingi konkreetne idee. Näiteks tsüklite õpetamist saab alustada üldisest while-tsüklist, et õpijad saaksid aru, kuidas täitmise ajal indekseid muudetakse ja tingimusi kontrollitakse. Filtris on praegu olemas kolm keeleprofiili, igaüks nendest lubab kasutada ainult teatud alamhulk Pythoni võimalustest.

Teine probleem seisneb selles, et Python on dünaamiline keel ega analüüsi koodi staatiliselt. See tähendab, et vigu avastatakse ainult käivitamise ajal. Iga trükkimisviga ja väärte tüüpide kasutamine funktsiooni argumentideks lõpeb kohe programmi täitmist, kaotades tulemusi. Baaskursustes tuleb niisuguseid olukordi vältida.

Filter loob Pythoni peale tugeva tüübisüsteemi ja kontrollib lähtekoodi tüübide kooskõla. See võimaldab vähemalt trüükivigadest ja enamasti valedest tüüpidest kasutajat informeerida enne täitmist.

Kui kasutatud elemendid on kõik lubatud ja tüübid langevad kokku, siis süsteem annab programmi tavalisele Pythoni interpretaatorile ja näitab kasutajale väljundit.