Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Calling Python from Prolog: A General Interface

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Lars Leyendecker

Beginn der Arbeit:	02. Mai 2022
Abgabe der Arbeit:	15. August 2022
Erstgutachter:	Prof. Dr. Michael Leuschel
Zweitgutachter:	Dr. John Witulski

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 15. August 2022

Lars Leyendecker

Abstract

Prolog used to be an essential programming language for AI research. However, it was replaced by other languages such as Python over time. Due to the shift in attention and resources, the Python module library today contains many efficient and thoroughly tested implementations of AI-concepts that are not available as a native Prolog implementation. Bringing these innovations to Prolog by directly translating them would be impractical. Additionally, the new code would need to be maintained to stay up to date, since many of the Python modules are in active development. Therefore, it would be desirable to access the capabilities of Python modules, without re-implementing them in Prolog.

In this work I present an interface between *SICStus* Prolog and Python named Prothon. It allows arbitrary function calls to be executed on a Python server from within a Prolog client. Additionally, Python objects can be returned to the client as references allowing them to be manipulated using the interface from within Prolog.

I investigated the usability and performance of the interface in two case studies. First, I examined the difference in runtime for an implementation of matrix multiplication between Prolog and Prothon. The results show that a computational speed improvement for numeric calculations can be achieved by using the interface, even though this could be further improved in the future. Second, I looked at the viability of utilizing the interface in a real use case. In the study machine learning models that where initially trained in Python were successfully loaded by Prothon and subsequently used for predictions. The pretrained decision tree models were saved and loaded using an available python module. Implementing and training those decision trees in Prolog directly would have been a significantly more complex and costly task.

Contents

1 Motivation **1**

2 Background **2**

2.1 Prolog . . . 2
2.1.1 Unification . . . 3
2.1.2 SLD Resolution . . . 4
2.1.3 Difference Lists . . . 5
2.1.4 Definite Clause Grammars . . . 5
2.2 Python . . . 7
2.2.1 Design . . . 7
2.2.2 Object Types . . . 7
2.2.3 NumPy . . . 8
2.2.4 SciPy . . . 8
2.2.5 Joblib . . . 9
2.3 Comparison Between Python and Prolog . . . 9
2.4 Sockets . . . 10

3 Implementation **10**

3.1 Communication Protocol . . . 10
3.2 Parsing . . . 12
3.3 Referencing and Dereferencing Objects . . . 12
3.4 Commands . . . 14
3.5 Concurrent Execution . . . 14
3.6 Garbage Collection . . . 15
3.7 Exceptions . . . 16

4 Case Studies **16**

4.1 Matrix Multiplication . . . 17
4.1.1 Comparison between Prothon and a Native Prolog Implementation . . . 17
4.1.2 Measuring the Impact of Individual API-Calls . . . 20
4.2 Using Pre-Generated Models . . . 22

5 Related Work **24**

6 Future Work **25**

6.1 Port to Other Versions 25
6.2 Garbage Collection 26
6.3 Security 26
6.4 Persistent Sessions 27
6.5 Dereferentiation Interface 27

7 Conclusion **27**

Appendix A Documentation **29**

A.1 Prothon Representation of Python Types 29
A.2 Public Interface 30

List of Figures **32**

List of Tables **32**

List of Algorithms **32**

List of Listings **32**

References **33**

1 Motivation

The logical programming language Prolog used to be essential in the field of artificial intelligence soon after its inception in 1972 [Bra12]. AI was a relatively new branch of research at that time and the dominance of Prolog in that field was only shared by Lisp [Lug01]. Due to its core design as a logical language it was and continues to be suited particularly well to transparently implement a way of reasoning within a certain domain. Together with its exceptional aptitude for natural language processing – the task Prolog was originally designed for – it was successfully used to implement systems that process human language, extract facts and use them to answer queries in human language as early as 1972 [CR96].

While Prolog has had a big impact on the transparent approach to AI called symbolic AI, the focus has shifted towards sub-symbolic and data-driven AI in recent years [KLB+22]. These approaches are often less transparent than symbolic AI, with some methods producing completely intransparent black boxes.

With the shift of attention also came a shift of resources, which ultimately led to other languages taking the spotlight. By now, many rich and sophisticated AI libraries have been developed for those languages and are continuously being worked on. One of the most popular is the general purpose and multi-paradigm language Python [VR+07].

Due to its popularity, machine learning has seen a lot of innovations over the past decade. Introducing these into the Prolog ecosystem would be desirable. However, rebuilding the code in Prolog has several disadvantages. Due to the large size of the Python library it would be ineffective to translate all the modules. Furthermore, the new code needs to be maintained to stay up to date with the state-of-the-art. Additionally, Machine learning algorithms apply many numerical computations, which Prolog does not excel at. Python, on the other hand, provides modules handling this task much more efficiently such as *NumPy* [HMvdW+20] and *SciPy* [VGO+20]. Instead, it would be favorable to allow access to highly tested and efficient implementations that are already available as Python modules.

For this purpose I will implement an interface between SICStus Prolog [CWA+22] and Python as a client-server architecture. This interface provides access to state-of-the-art machine learning libraries and allows Prolog to benefit from any future updates or innovations in Python. The server will be running in Python and accepting connections from clients over a TCP-Socket. A Prolog client can connect to the server and execute Python functions.

Apart from providing access to readily available technologies to Prolog, the interface can also be used to create new software by utilizing both languages' strengths when developing new AI technologies. Prolog can, for example, be used to preprocess data in order to prepare it for usage in one of the many data-driven AI approaches [KLB+22]. A project focusing on natural language processing could take advantage of both Prolog's definite clause grammar feature to easily implement a formal grammar and the efficient numerical

vector calculation of *NumPy* and *SciPy*. It also enables developers to write Python code specifically designed to be run with the interface to outsource parts of a Prolog program because it runs more efficiently in Python or is simply less complicated and therefore more practical to be implemented in that language.

2 Background

In the following, I will present the necessary background knowledge for this thesis and the implementation of Prothon. This includes bridging the two programming languages Prolog and Python, Python libraries such as *NumPy* and sockets used as the communication channel between client and server.

2.1 Prolog

The Prolog [CR96] programming language emerged in 1972 out of a project originally intended to achieve natural language processing. Today different implementations of Prolog exist. In 1985 the *Swedish Institute for Computer Science* (SICS) was founded and begun work towards a Prolog engine [CM12].

Prolog is a symbolic logical programming language [Bra12]. It is a dynamic and interpreted language, although it can also be complied to virtual machine code. It uses the declarative programming paradigm, which differs from the more well known paradigm of procedural programming in how solutions to problems are described. In Prolog one describes how data is connected and how a solution would look like instead of providing a detailed step by step algorithm describing how to calculate a solution. Then, it is the Prolog engine's task to find solutions for queries using an efficient resolution algorithm embedded in the Prolog system.

Recursion is often used to describe the relation between data in Prolog. The simplest recursive definition requires two cases. First, a base case that expresses the relation between the data in a specific simple case. It is commonly the case in which the data is in its simplest possible form. This is often an empty data structure, e.g., an empty list. A base case could therefore be *two empty lists are equal*. Second, a general recursive case is described. Here the relation between data is defined in terms of the relation of a simpler version of the data. How exactly the data is simplified depends on the relation that is being described and the base case's definition. An example of a recursive case is *two lists are equal if their first elements are equal and the rests of the lists are equal*. Listing 1 shows how such a recursive definition can be implemented in Prolog. Multiple base and recursive cases can be used together.

Because recursion is used often, Prolog is optimized to handle it efficiently. Tail-recursion optimization allows the interpreter to reclaim the program stack, instead of writing an entirely new stack frame for every recursive call, as long as the recursion occurs at the very

Listing 1: The equality of two lists is described recursively in Prolog.

```
1: % two lists are equal
2: % - if they are both empty
3: equals([], []).
4: % - if their first elements are equal
5: %   and the rests of the lists are equal
6: equals([H|T1], [H|T2]) :- equals(T1, T2).
```

end of the description. This saves both memory space and computing time.

In Prolog there is only one datatype as everything is a term [CWA+22, Chapter 4.1.2]. A term can be an atom, a number, a variable or a compound term. An atom such as `empty` is the simplest kind of term as it is just a name without any implicit meaning. A number can be an integer or a float. Variables represent arbitrary terms and are always immutable. Thus, after a term was assigned to them, it is not possible to change it anymore. They always start with an upper case letter or underscore to distinguish them from atoms. A compound term combines multiple terms into one, by using an atom as a functor and several other terms as arguments [CWA+22, Chapter 4.1.3]. For example `node(empty, 42, empty)`. The number of arguments of a given functor is called its arity. If compound terms with the same functor need to be differentiated, they are appended by a forward slash and their arity, e.g., `empty/0` or `node/3`.

Every Prolog program consists of a number of clauses consisting of a head and a body as shown in Listing 1. The head of a clause is a compound term whose functor is the clause's name. The body consists of other compound terms – usually separated by commas or semicolons, meaning logical AND and logical OR respectively. More complex logical constructs such as if-then-else and negation do also exist. The head of the clause is considered true if the body is true. The body might be empty in which case the clause is a fact and always considered true as can be seen in the first clause in Listing 1.

In Prolog everything that is not explicitly true is considered false. A proof by contradiction is used to find out if a query is true. This means negating the query and trying to find a contradiction. If a contradiction is found, then the query must be true. If it is not possible to find one, then it must be false. A query consisting of a conjunction of terms is therefore negated to form a disjunction of negated terms, which is called a goal clause. All the other clauses in the Prolog program are definite clauses, meaning they are disjunctions containing exactly one term – specifically the head – that is not negated.

2.1.1 Unification

Prolog uses unification in its resolution process. Two terms can be unified when they are either identical or if "the variables in both terms can be instantiated to objects in such a way

that after the substitution of variables by these objects the terms become identical" [Bra12, p. 39].

This means that constants such as atoms and numbers are unified if they are identical and compound terms are unified if they share the same functor and arity, and all of their components unify recursively [Bra12].

Unification normally includes an *occurs check*, which checks if a variable is occurring inside a term that is to be unified with that variable [CWA+22, Chapter 4.2.7]. If that is the case, unification should not be successful. In Prolog however this check is omitted for performance reasons, because it would make the unification an operation with linear instead of constant performance. Unifying a variable with a term containing that variable results in a cyclic term. If those are not handled accordingly, they can cause infinite loops in some cases.

2.1.2 SLD Resolution

Selective linear resolution with definite clauses (SLD resolution) [KK71] is used to derive new goal clauses until either no more derivatives can be formed or if the derived goal clause is empty, which means a contradiction has been found.

The query is first negated to obtain a goal clause. The Prolog interpreter then iteratively applies Boolean resolution with unification. By using Boolean resolution one is able to deduce $(\alpha \vee \gamma)$ from $(\alpha \vee \beta) \wedge (\neg\beta \vee \gamma)$, when α, β and γ are terms. In each step of the resolution a negated term of the goal clause is unified with the positive term of a definite clause. Since all other terms of the definite clause are negated, it is ensured that the deduced clause must, again, be a goal clause. Therefore, another step of the resolution can be performed.

Selecting which term is used to derive a new goal is not deterministic as there could be multiple clauses that could be used in the derivation step. In this case one has to be chosen. In Prolog the clauses are chosen top to bottom. For every such choice a choice point is created. This can be visualized by a tree branching out every time a choice point is encountered. This tree is searched for solutions depth first, which means that it will first exhaustively traverse down one path of the tree. If the resolution fails, Prolog will jump back to the last choice point and try the resolution again with a different clause. Backtracking causes all the variables that were unified after the choice point to be unbound. A Prolog interpreter is able to find all solutions for a query by exhaustively evaluating all choice points.

The special operator `!` – called *cut* – can be used to prevent backtracking and thus reduce the search space. When this term is encountered, the subtrees of the previous choice points get cut away and only the branch that is currently being explored is left. This is a local operation affecting any choice points up to – but including – the predicate that the cut is

in. Future choice points are not affected. Well-placed cuts can improve the efficiency of predicates, for example, by preventing the engine to search for alternative solutions that do not exist. Since the cut is not a purely logical operator, any predicate that makes use of it can not be purely declarative.

2.1.3 Difference Lists

The task of generating or manipulating data structures often involves the building of lists. While prepending an element to a list takes a constant amount of time and therefore is in $\mathcal{O}(1)$, appending an element to a list requires traversing the list fully. Similarly, appending one list to another requires the first (but not the second) list to be traversed completely. Thus, the time needed to append one or more elements to a list with n elements is $\mathcal{O}(n)$.

Special lists – called difference lists – can be appended to in constant time. They are similar in idea to saving a pointer or reference to the last element of a singly linked list in other programming languages. In Prolog this is achieved by using unification, the `[Head|Tail]`-Notation and the fact that structures do not need to be completely defined right away.

A difference list is a list with a variable in its tail. They can, for example, be stored as a term `DL-T`, where `T` is the variable in the tail of `DL`. For instance, `DL-T = [1, 2, 3|T]-T` is a valid difference list.

To add an element the variable `T` needs to be unified with a list containing the element and a new tail, say `T2`, which would result in the difference list `L-T2` as can be seen in Listing 2. It can also be seen in the listing that two difference lists can be appended in constant time. Finally, they can be turned into a regular list by unifying the tail with an empty list.

2.1.4 Definite Clause Grammars

An outstanding feature of Prolog called definite clause grammars (DCGs) enable the manipulation of lists with easily definable grammar rules [CWA+22, Chapter 4.14]. This especially allows parsing strings as they are represented as lists of characters. DCG clauses look similar to regular Prolog clauses and also only allow one term as the head. Their body consists of terminating symbols, which are atoms or arbitrary lists not containing any variables, and non-terminating symbols, which are other grammar or regular Prolog clauses, separated by a comma. DCGs can be used to implement context free grammars.

When encountering these clauses, the Prolog interpreter uses its term expander to transform them into regular clauses with two additional parameters corresponding to a difference list, as can be seen in Listing 3. If the grammar is used to parse a string or list, the first variable contains all the symbols or elements that still need to be consumed before the rule was applied, while the tail contains all the symbols or elements that are still not consumed

Listing 2: Difference lists are used to append to lists in constant time.

```
1:  % Appends 'Element' to the difference list L1-T
2:  % resulting in the DL L2-T2
3:  add_element(L1-T1, Element, L2-T2) :-
4:  T1 = [Element|T2],
5:  L2 = L1.
6:  % This can also be written as
7:  add_element_compact(L-[Element|T], Element, L-T).
8:
9:  % Appends the difference list L2-T2 to the DL L1-T1
10: % resulting in the DL L3-T3
11: append_dl(L1-T1, L2-T2, L3-T3) :-
12: T1 = L2,
13: L3 = L1,
14: T3 = T2.
15: % This can also be written as
16: append_dl_compact(L1-L2, L2-T2, L1-T2).
17:
18: % Finalizes the difference list by unifying the tail
19: % with an empty list
20: dl_to_list(L-[], L).
```

after the rule was applied. In the case that a string or list is generated by the grammar it is in fact not different, even though the contents of those lists are partially unknown at the time. The first variable holds everything yet to be generated before the rule and the second holds everything that will be generated after the rule is applied. What was generated is appended to the first list and its new tail is unified with the second list. By using the tail of one rule as the list in the next rule, the lists get unified into one list with constant time complexity in each step.

Listing 3: The term expander translates grammar clauses to regular Prolog clauses.

```
1: reverse([H|T]) --> reverse(T), [H].
2: reverse([]) --> [].
3:
4: % gets translated by the term expander to
5: reverse([H|T], L0, L2) :-
6:     reverse(T, L0, L1)
7:     L1 = [H|L2].
8: reverse([], L0, L0).
```

2.2 Python

Python [VR+07] is an interpreted general purpose programming language authored by Guido van Rossum and was first released in 1991. Its core design philosophies include a focus on code readability and simplicity.

Information in this section is taken from the official documentation for Python 3.10.6 [PD22] unless stated otherwise.

2.2.1 Design

One part of the strategy to achieve code readability is the use of significant indentation, where indentation is not optional as it is in most programming languages, but defines code blocks [vRD10]. This enforces indentation – improving readability – while also eliminating the needs for brackets. Hard to spot programmer mistakes where indentation does not match the actual semantic structure of a program can therefore not occur in Python since the indentation itself defines the semantic structure.

As a multi-paradigm language it fully supports both procedural and object-oriented programming. It also allows functional programming, by way of writing functions whose output only depends on their input and using language features such as iterators, generators and higher order functions, which accept functions as their inputs.

Memory is handled entirely by the Python memory manager without any control by the user. The memory manager uses reference counting to determine if objects can be freed and uses an optional garbage collector to discover objects only kept alive due to reference cycles, which the memory manager can not detect on its own.

If an error occurs during execution the program is not immediately terminated, but instead an exception is raised. This exception can then be caught and appropriate actions can be taken to deal with that problem. If an exception is not immediately caught where it occurred, it will propagate up through the call-stack until it is either caught somewhere or causes the Python runtime to terminate the program if it was not caught at all. Exceptions do not necessarily need to indicate an error, but can also be used for control flow. For example, if new values are repeatedly requested, until an exception indicates that no more values can be served. They are distinguished by their type and can be raised manually at any point in the code.

2.2.2 Object Types

Everything is an object in Python including simple built-in types such as integers (with arbitrary precision), floats and complex numbers, but notably also functions.

It also comes with multiple built-in types of collections. Lists are mutable arrays while tuples are immutable. Sets are a duplicate free and unordered collections similar to sets in mathematics and also provide the common set operators. Dictionaries define a mapping where each value is stored under a key.

Python has an extended syntax for the usage and processing of lists making it more compelling for data science. Apart from the syntax for list indexing that many languages provide, there is also dedicated syntax for list slicing where the start and end index can be specified either relative to the start or end of the array. This is expanded on in the *NumPy* package.

2.2.3 NumPy

The foundation of many of the scientific modules is the library *NumPy* [HMvdW+20], which adds a high performance array object alongside many mathematical functions to be used with these arrays [vdWCV11]. It enables highly efficient vectorized computing, while doing so with a concise and intuitive syntax that often resembles mathematics. This is supported by Python's ability to modify build in operators such as the plus sign to perform vectorized addition between *NumPy* arrays.

The vectorized functions are implemented in C++ or Fortran and often use preexisting routines that are already proven to be efficient prior to their use within *NumPy* [vdWCV11]. This practice makes numerical computation fast regardless of the fact that Python itself is an interpreted language, since most of the work is done in the complied code instead of in Python.

2.2.4 SciPy

Build on top of *NumPy* is the module *SciPy* [VGO+20] including algorithms for many mathematical problems such as integration, differential equations, interpolation and eigenvalue problems. It is itself divided into several sub-packages one of them being *scikit*, which is developed outside of *SciPy*. It is meant for more experimental submodules, which would not be suited in *SciPy* due to their lack of the required stability.

Scikit itself contains several submodules, such as *scikit-learn* [PVG+11] (also referred to as *sklearn*), a module containing many machine learning algorithms concerning classification, clustering or regression. It also includes a lot of sample datasets that can be used to test out a wide variety of machine learning applications.

2.2.5 Joblib

Fitting a machine learning model can require a substantial amount of computation, especially for larger datasets. Persisting the trained models to disk is therefore an essential task in order to avoid running the same computations multiple times. The module *Joblib* [JD21] is able to serialize and store any Python object in a file. Stored objects can be loaded to reconstruct the saved object entirely.

Storing objects on disk enables them to be saved and loaded in different sessions of a single Python program, as well as sharing objects between entirely different programs. Furthermore, a model trained in a Python program could be loaded by the Prothon server. Thus, making the model accessible by the Prothon client.

2.3 Comparison Between Python and Prolog

A comparison of the two languages is advantageous in order to point out similarities and differences:

- Both languages are interpreted and able to be run inside a Read Eval Print Loop (REPL). The Python server for Prothon could work similarly to a REPL by receiving queries, which are parsed, evaluated and answered before waiting for the next query.

- Python and Prolog are both dynamically typed. This allows the types of sent and received data to be inferred at runtime. Because there is no need to explicitly type cast to conform to function signatures, there is also no need for the server to communicate any function signatures to the client or for any such information to be present beforehand.

- Another similarity is that in both languages all data has one supertype. While in Prolog everything is a term, in Python everything is an object. This allows both languages to support introspection and reflection, which is the examination and manipulation of all of their data and makes meta programming possible. In Python this can, for instance, be achieved with annotations manipulating functions, while in Prolog the term expander can be used to manipulate terms.

- Both languages handle errors by way of throwing and catching exceptions. This means that exceptions that occur on the server, but are not caught on the server, can be passed down through the API to then be thrown in the client. They can then either be caught and dealt with or cause the client to abort execution if they remain unhandled.

2.4 Sockets

An appropriate channel is needed in order to establish a connection between client and server. For this purpose I will use sockets.

A socket is a mean of communication between two processes. Different types of sockets exist. Unix domain sockets can be used for communication between two processes that run on the same operating system. TCP/IP sockets enable the communication of processes that might run on different operating systems and machines connected by a computer network, but it can also be used for communication local to an operating system, similar to Unix domain sockets.

At the high-level a TCP/IP socket is a bidirectional byte or text-stream to which both ends can write and read messages. These streams are not partitioned or otherwise organized into parts, but instead determining the beginning and end of an individual message is a responsibility of the communicating parties. This is usually done by agreeing upon a protocol governing the communication.

3 Implementation

The original motivation for the interface was to allow the use of one or more of Python's artificial intelligence modules from within Prolog. When thinking about the approach, the simpler and more efficient task seemed to be solving a general case instead. This would result in a simpler and more maintainable program capable of supporting more modules. I concluded that instead of exposing some Python modules through a specialized interface it should simply be possible to execute arbitrary function calls on a server that would store generated objects and would return references to those for use in future calls. The usage of any arbitrary artificial intelligence module is then also solved as a special case of this more general approach.

3.1 Communication Protocol

To establish a communication between the client and server, a TCP/IP socket is used in text mode. Server and client communicate over a simple protocol where the client sends out queries and the server only sends an answer when this is explicitly requested by the client.

Both query and response have a header containing 10 characters which are a base 10 representation of the number of bytes of the message excluding the header itself. This allows the last argument to not be delimited and eliminates the need for extensive manual bracket and delimiter matching to identify the end of the message.

Every query starts with the name of the requested operation followed by a comma and

then a comma separated list of arguments for that specific operation. For example, the `import` command has the two arguments *module* and *alias*. Therefore, the query to import *NumPy* with the alias *np* would be `0000000015import,numpy,np`.

An argument can be unquoted text, an integer or any Python constant, object or arbitrarily nested collection, depending on the command. I chose to only allow commas in the last argument of any query. This allows for a clear splitting of queries into parts. As a consequence only the last argument may be a Python collection.

Constants and collections will be transferred in literal Python format, that is in a way that it could be typed literally into the interpreter. This allows the parsing to be handled entirely by the Python interpreter itself, eliminating the need of manual parsing completely.

Object references are transferred as `py[ID]`, where `ID` is the integer constant that identifies the object on the server. The dictionary containing the reference files on the server is aliased as `py[ID]`, which is the literal way to get the object with the specified `ID` from that dictionary. The identifier can be parsed and evaluated by the server as is. While this only plays a role in the communication from the client to the server, the same format is used for communication in the opposite direction to keep the protocol simple.

Strings are delimited by double quotes. Double quotes in strings thus need to be escaped using a backslash. To avoid further problems that would arise from a backslash at the very end of the string effectively escaping the closing double quote, backslashes themselves also need to be escaped.

Answers from the server to the client always start with a digit denoting either success or failure. This is not part of the header, but part of the message after the header.

The response to a successful query starts with a `0` followed by exactly one Python object, constant or arbitrarily nested collection. Constants and collections are also sent in their literal Python representation while the same rules for object references and strings apply as they do in queries.

If the query was not successful, the answer starts with a `1` that is followed by a list containing two strings. The first string is the type of the exception that was thrown, and the second is the string representation of the exception usually providing more information on the cause of the problem.

The choice to use literal Python as the syntax for both query and answer has two major benefits. First, it is completely independent of Prolog allowing the server to be reused in the future with any client written in any other language. Second, it puts all the work in terms of manual parsing onto the Prolog side, which is generally better equipped to do that task, for instance, by providing definite clause grammars.

3.2 Parsing

Parsing queries on the server can be done in three simple steps. In the first step the literal object is parsed into an abstract syntax tree by the Python interpreter. The AST is then compiled and evaluated by the interpreter into an actual Python object. Only replacing the object references sent by the client into actual reference on the server requires the implementation of custom code. This is done by creating a subclass of `NodeTransformer`, which replaces every name in the AST with the corresponding object in the server's dictionary.

Traversing an AST has the benefit that it can be used for security related tasks as well such as checking that no other names get referenced or that no function calls are executed within that query. A downside is that traversing the entire AST introduces a significant amount of additional overhead.

For that reason I will use an alternative implementation that writes the references as complete dictionary references that can directly be interpreted by Python. This eliminates the need to go over the AST completely. To keep the amount of bytes required to transfer one object reference minimal, a short alias is used to access the dictionary.

How an answer is generated by the server depends on the level of dereferencing that is requested. If no dereferencing is necessary the object is simply added to the server's reference dictionary and a string representation of this reference is returned.

Detrimental to derefenrencing objects is a prior determination of the object's type. In case of a string any contained double quotes and backslashes are escaped and the string is wrapped in double quotes. Floats, integers and Boolean objects are just returned as their string representation. In the case of a collection such as a list, tuple or dictionary, the appropriate type of brackets is used and their elements are separated by comma. The elements themselves are either added to the reference dictionary and replaced by a string representation of the reference or recursively turned into string representations of their values depending on the desired depth of dereferentiation.

The client uses a definite clause grammar for parsing. The same DCG is used to generate queries for the server and to parse answers from the server. Hence, it is translating between terms and strings in both direction. This is possible due to the formatting of Python literals being identical in queries and answers. Only a few grammar rules are unidirectional, e.g., the translation of numbers.

3.3 Referencing and Dereferencing Objects

A core element of the API is the way object references are handled and communicated between the server and the client. When an object reference should be transferred to the client, the object is added to a dictionary on the server with the return value of the `id()`

function as its key. This identifier is the integer representation of its identity in Python. For this purpose it is guaranteed to be both constant and unique for the entire lifetime of the object, which of course also makes it perfectly suitable to identify the object in a dictionary. Instead of the object, its identifier is then sent to the client. When using the object as an argument or as part of an argument, or when calling a method on an object, this reference can be passed back to the server, where it will be replaced by the actual object.

After passing data back and forth in a series of interface calls, there is an obvious need to directly access the generated or manipulated data. In order to be usable in Prolog, the data is required to be in a native format and must not contain any references to Python objects. However, intermediate data – only used as arguments in additional interface calls – can be either constants or references.

My first thought was to reply with actual data opposed to references whenever possible. This means that every built-in type such as integers, floats and strings, but also collections such as lists, tuples and dictionaries are always transferred back as a constant. While this provides the feeling of a more seamless experience when using the API, the drawbacks are apparent. Data might be returned from Python only to be used as an argument in another call. First, this data is transferred over the socket twice needlessly. Second, it needs to be processed a total of four times. When handling large lists or arrays which, for instance, is a common occurrence for machine learning applications, this can induce a significant amount of overhead. This makes it apparent that all collections should generally be transferred as a reference rather than a constant.

While for primitive types this overhead is not a big concern, I decided to avoid exceptions wherever possible and sensible in favor of a clean design. Since everything is an object in Python, everything will be treated as such and returned to the client as a reference. The only exception is the constant `None` which, while technically being an object too, implies the absence of an object as it is a reference to nothing. For that reason it is sensible to return it as a constant rather than a reference.

Transferring everything as a reference leaves the problem that data needs to be made available to the client if desired. For this purpose, a predicate to dereference an object is required. In this context, one question is how deep a potentially nested collection should be dereferenced. While it will often be desired to fully acquire the values of arbitrarily nested collections exhaustively, always doing so will again lead to unnecessary overhead as laid out previously. In some cases this can even be counterproductive, e.g., splitting a tuple of arrays. Since the representations sent to the client would be a copy instead of a reference, it would not be possible to manipulate the original arrays on the server.

Therefore, a predicate capable of dereferencing exactly one level of depth is required. While the option of being able to dereference to an arbitrary level of nesting exactly would not be redundant, I decided against providing this feature for the sake of a simpler interface. Accessing the data one step at a time also encourages lazy evaluation, where the data is

only dereferenced when it is really needed, instead of eagerly acquiring values that possibly will never be used.

Dereferencing is currently supported for the built-in types integer, float, string, tuple, list and dictionary, as well as for *Numpy*-arrays and *Numpy* generic values. Other types have to be dereferenced by either calling appropriate functions that return a representation of the object that can be dereferenced, or by extending the parser with the appropriate dereferentiation logic. In the future this might also be done by a specialized interface.

In case of an exception causing the query to fail, the type and string representation of the exception is always transferred as constants and never as a reference to allow the client to throw that exception immediately without having to request the details of the exception separately.

3.4 Commands

The commands available by the server fall into two categories. The queries, which all push one object onto the answer-stack while possibly manipulating the server's state and the `fetch` command, which retrieves an object from the top of the stack.

The result generated by any of the former commands, whether it is an exception or an object, is not immediately sent back to the client as shown in Figure 1. Instead, it is pushed onto a stack and only when explicitly requested – by use of the `fetch` command – the answer on top of the stack is sent as an object reference or literal depending on the request.

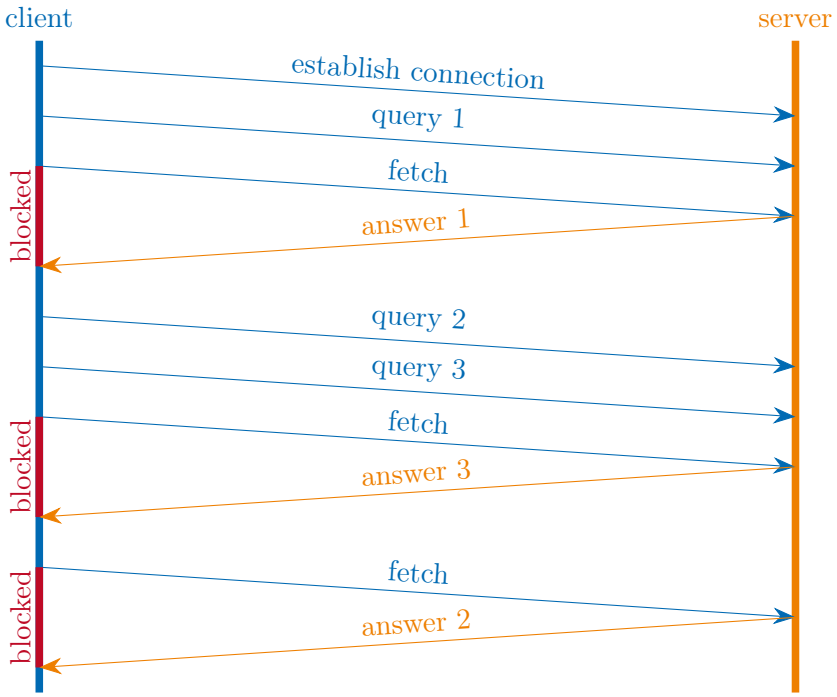
This decoupling of query and answer makes a concurrent execution possible. Due to the queries not having to block until an answer is received, other code can be executed while the query is processed on the server. The fetch command is the only blocking command. It is blocking execution until an answer is received as shown in Figure 1.

Furthermore, the decoupling keeps the logic simple as each command retains one task. Implementing the commands in a way that every command could also directly elicit a response would only increase complexity, without providing any significant benefit. Whether an answer is to be sent and how deep the object is to be dereferenced would need to be indicated for every query. Therefore, an additional parameter would always be required. However, since a command to simply fetch an answer from the stack is needed in any case to enable concurrent execution of queries by splitting query and answer, it would just be additional complexity that is capable of the same tasks.

3.5 Concurrent Execution

Almost all predicates that involve communication with the server contain a `fetch` command and therefore block until an answer is received. Contrarily, the predicates `pycall/2`,

Figure 1: The interface uses a client-server architecture. Answers are only sent by the server when they are explicitly fetched by the client.



`pycall/3`, `pymethod/3` and `pymethod/4` only send out a query but do not fetch the answer right away. Since the answer is pushed onto the answer-stack, it can be fetched manually at a later time using `pyanswer/1` or `pyanswer/2`. This enables the programmer to perform the blocking call to fetch the answer of a possibly time-intensive task only when it is necessary. This allows for execution of other code in the meantime. Importantly, objects stored on the answer-stack are kept in memory until fetched. Therefore, discarding objects by never fetching them causes the server's memory to fill up. Objects should always be discarded using the provided garbage collection.

3.6 Garbage Collection

Without garbage collection all objects sent to the server would be kept in memory indefinitely. After continued use the server would eventually deplete all of its available memory, causing either the memory getting written to disk – thus slowing the process down – or the refusal of the operating system to give out new memory blocks causing the server to crash. To prevent this, the client needs to inform the server which objects are not needed anymore or alternatively which objects are the only ones that are still needed. It is the programmer's task to do this manually. For this purpose there are two different server commands and respective Prolog predicates.

- `pygarbage/1` takes a list of object references and explicitly deletes those from the server.

- `pykeep/1` also takes a list of object references, but deletes every reference from the server that is not in this list. This predicate is provided for convenience in the use case that after manipulating data and generating a lot of intermediate objects only a few of the objects are actually required. It is important to understand that this is a global operation. It must be known exactly which objects are still needed globally in the program or else objects that are still needed elsewhere will get deleted. This predicate – or more specifically its corresponding command in the server – could be used in the future for more automatic garbage collection, by figuring out which references are still accessible within Prolog and discarding the rest.

It is important to note that the references only get removed from the dictionary on the server and it is up to the Python memory management and garbage collector to remove the objects from memory if warranted. This means that objects can always be safely passed to the garbage collector even if they might still be referenced inside other objects.

3.7 Exceptions

In the case that an exception was thrown and never caught on the server, it will be transferred to the client in place of an answer. This can either be an exception that occurred in a called function or in the internal server logic. An exception due to internal sever logic should however never be raised due to a bug in the server, but only due to misuse of the interface. For instance, if an unknown function is called.

When the client detects that the answer is an exception, it will parse its type and message. The exception is then thrown in the client where it can be handled by the caller. The functor of every exception thrown by Prothon is `prothon_exception/2`. The arguments are the type and string representation of the exception respectively as atoms. For instance, `prothon_exception('ModuleNotFoundError', 'No module named \'numpi\'')`.

4 Case Studies

To evaluate the performance of the interface I have chosen to conduct two case studies focusing on speed and usability, furthermore taking into account both the performance of the Python server and the overhead introduced by the interface.

4.1 Matrix Multiplication

Matrix multiplication requires many numeric calculations. Multiplying two $N \times N$ matrices has a complexity of $\mathcal{O}(N^3)$ and should therefore be suited to study the performance of the interface and its possible speedup.

I will compare the different approaches to the implementation of object referencing as outlined in Section 3.2. In the first approach references are implemented by encoding them directly when generating the query in Prolog. In the second approach they are implemented by using a placeholder that will be replaced with the references by traversing over the abstract syntax tree generated during parsing on the server.

4.1.1 Comparison between Prothon and a Native Prolog Implementation

One distinct use-case of the interface is to speed up complex algebraic calculations. The Prolog interpreter is expected to perform these much slower than a sophisticated Python library designed for that type of calculation. An example for such a calculation is the matrix multiplication. The study tries to answer two questions.

1. How does Prothon perform against a native approach when multiplying two matrices?
2. How much of the interface's performance depends on the overhead caused by the interface opposed to the calculation itself?

I expect the native predicate to have a better performance with small matrices due to the overhead introduced by the interface. Since the matrix multiplication has a complexity of $\mathcal{O}(N^3)$ for two matrices of size $N \times N$, I also expect the time necessary to perform the calculation to outgrow the interface's overhead and become the dominating factor quickly.

The benchmarks were carried out with square matrices in a range of different sizes with randomly generated entries. Scaling the sizes linearly would result in a resolution that would either be too dense for higher values or too thin for lower values. Instead, the sizes are distributed in a logarithmic scale. For each size one pair of matrices containing only integers and one pair containing only floating point numbers is generated. These pairs are then each multiplied once using a native Prolog implementation and once with Prothon calls that use *NumPy* for the calculations. Both predicates performing the multiplications have the same inputs and outputs. This makes them completely interchangeable, with the only difference being their execution speed. The described benchmark will be carried out once for the implementation in which the AST gets modified and once for the implementation in which the references are written directly into the query.

The benchmarks are carried out using the client. First, a list of 100 logarithmically spaced numbers is generated between 10 and 1000 using the interface and the `lnspace` function

Listing 4: The API is used to implement matrix multiplication.

```
1: pymatmul_steps(A, B, C) :-
2:     pycall('np.array', [A], dict([]), NpA),
3:     pycall('np.array', [B], dict([]), NpB),
4:     pycall('np.matmul', [NpA, NpB], dict([]), NpC),
5:     pyderefall(NpC, C),
6:     pygarbage([NpA, NpB, NpC]).
```

Listing 5: The metapredicate 'time' is used to measure execution times.

```
1: time(Pred, Time) :-
2:     statistics(walltime, [Start, _]),
3:     call(Pred),
4:     statistics(walltime, [End, _]),
5:     Time is End - Start.
```

of *NumPy*. This function generates a list of numbers in which each element multiplied by the same constant factor equals the next element. These numbers are then rounded down to the nearest integer and used as the matrix size. For each size, a total number of four matrices are randomly generated using the `library(random)` of *SICStus* Prolog. Two matrices contain integers and two containing floating point numbers in the interval $[-1000, 1000)$. Each of those pairs is used once for the native and once for the API driven matrix multiplication. The benchmarks are performed five times and the average of the measured times is calculated in order to minimize errors.

The interface-driven predicate (see Listing 4) first transfers each matrix – being a list of lists of numbers – to the server in a separate call to the *NumPy* function `np.array`. The function creates a *NumPy*-array from the list and returns it to the client. The references to these arrays are then passed as arguments in a call to the function `np.matmul`, which multiplies the two matrices and returns the result as an array. This array is then dereferenced completely to obtain the result of the matrix multiplication in the same format as the matrices were input into the predicate. Finally the three object references are passed to the server's garbage collector.

Each call is timed by the meta predicate `time`, by calculating the difference in elapsed time before and after the call in milliseconds as can be seen in Listing 5.

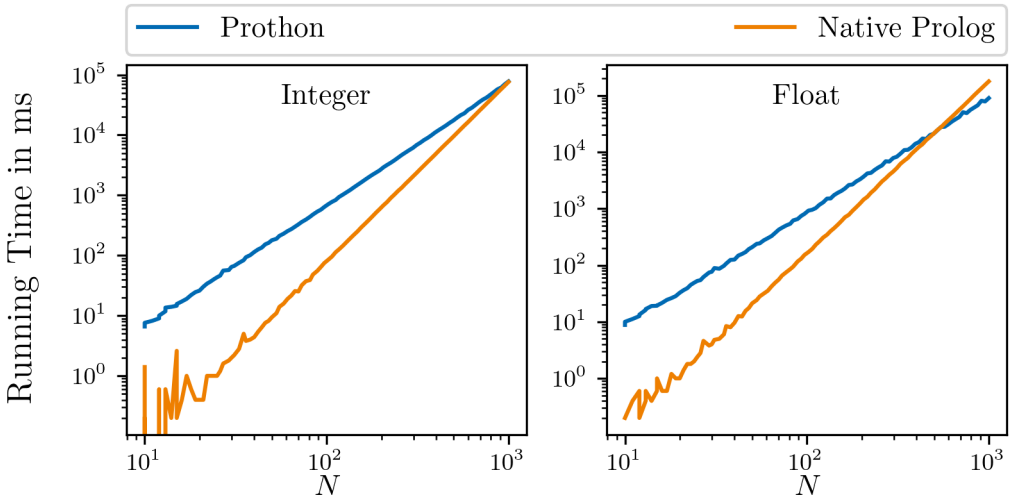


Figure 2: A graph depicting the execution time of multiplying two $N \times N$ matrices while traversing the AST.

Figures 2 and 3 show the resulting times [1] [2] in a log-log plot. While it takes around 550ms for two 100×100 matrices filled with integers to be multiplied with direct referencing, it took around 700ms when traversing the AST. Therefore, a noticeable speed gain was achieved and the assumption that traversing the AST introduces significant overhead is proved correct.

As seen in Figure 3 after a certain threshold the interface call is more efficient than a native implementation as was expected. For the implementation with direct referencing, this threshold is at a matrix size of around 800×800 for matrices containing integers. The threshold for floating point numbers occurs for matrices with a size of around 400×400

Although such a threshold will always exist, 400×400 matrices are relatively large. This raises the question of how much overhead is still slowing down the interface and how much more optimizing could possibly be achieved.

The methods of timing are not perfect. Since the elapsing of real time is measured, other processes running on the machine can influence the times. To minimize that factor the benchmark tests were performed with only the client and server running. Additionally, the measurements were performed 5 times before calculating the mean values to further reduce

[1] Direct referencing:
https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/lars-leyendecker/-/blob/f6b5de8002c23133200a4693ae6d69393367ec48/csv/benchmark_matrix_compare_direct_referencing/benchmark_matrix_compare_direct_referencing.zip

[2] Traversing the AST:
https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/lars-leyendecker/-/blob/f6b5de8002c23133200a4693ae6d69393367ec48/csv/benchmark_matrix_compare_traversing_ast/benchmark_matrix_compare_traversing_ast.zip

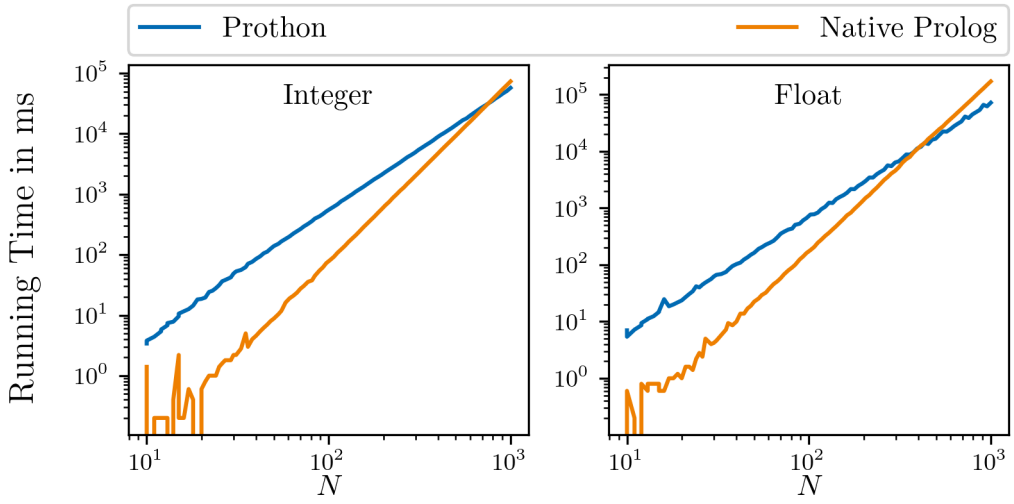


Figure 3: A graph depicting the execution time of multiplying two $N \times N$ matrices while encoding the references directly.

the error.

Since the times were measured in milliseconds, the inaccuracy is high when benchmarking small matrices. As the matrices grow bigger the relative significance of this error is reduced.

4.1.2 Measuring the Impact of Individual API-Calls

In order to find an answer to the open questions mentioned above, I will take a closer look into the performance of the predicate that performs matrix multiplication via the interface. Thereby, I will determine whether the bottleneck is the calculation itself or the overhead introduced by the interface.

For this study I will again look at the predicate in Listing 4, but instead of comparing it to a native implementation, the focus is on observing the relative weight of each call. The first two calls are transferring all the data to the server. The call to perform the calculation itself takes two references as arguments and thus does require only minimal parsing on both ends. The time impact of the calculation can thus be measured in isolation. The call to `derefall` is a good benchmark for the performance impact of encoding on the server side and parsing in Prolog. Finally, the garbage collection should only have a minor impact on the overall runtime.

In summary, the predicate is suited to measure the time impact of all three major tasks in isolation. The three parts are: transferring the data to the server (once for each matrix), performing the matrix multiplication on two arrays on the server and fetching the result as a native Prolog list.

To get the individual times, we use a modified version in which every Prothon-call is simply wrapped with the meta predicate `time` (Listing 5) and the measured times are written to disk. The benchmarks are performed over a list of approximately logarithmically spaced integers and matrices generated with random entries for each size as described in Section 4.1.1. For simplicity, we will only use matrices that have integers as their entries in this study.

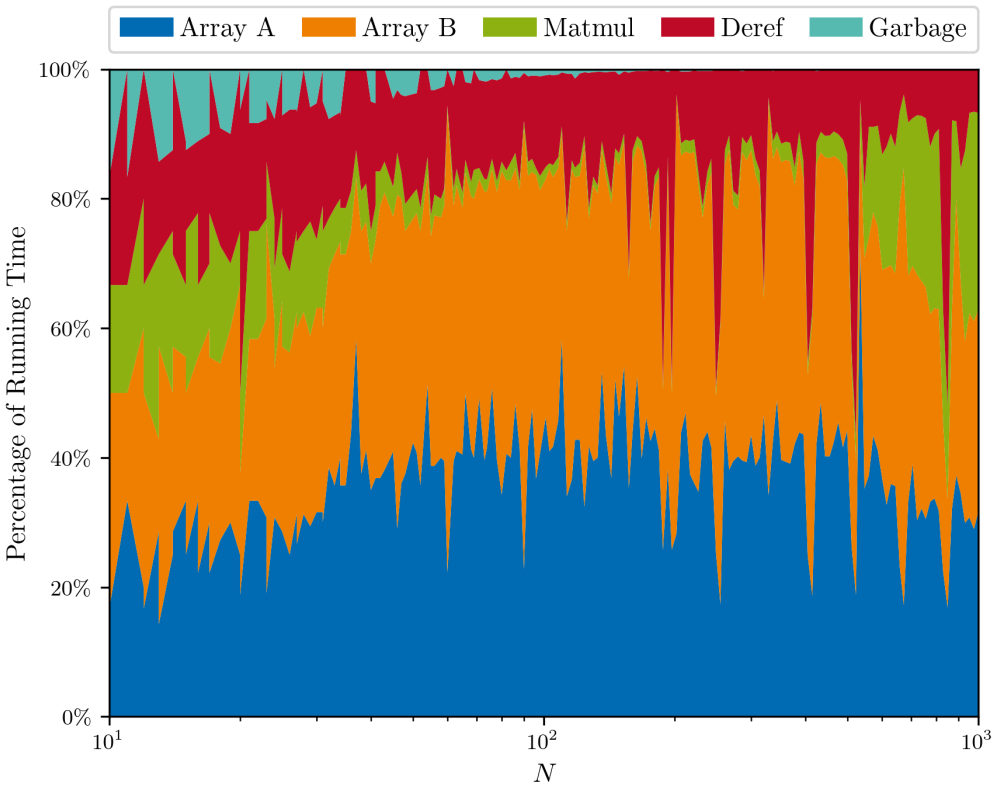


Figure 4: A graph depicting the relative performance impact of individual interface-calls when multiplying two $N \times N$ matrices.

The relative impact of the resulting times [3] was calculated and plotted in Figure 4.

Even without traversing over the AST transferring the two matrices to the server still accounts for $50 - 80\%$ of the running time in the range of matrices between 10×10 and 1000×1000.

Interestingly, the impact of the matrix multiplication peaks at the beginning, but starts to decrease first until eventually starting to increase again. While this might be unexpected, the reason for the early peak is that the runtime of the calculation includes some of the

[3] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/lars-leyendecker/-/blob/f6b5de8002c23133200a4693ae6d69393367ec48/csv/benchmark_matrix_dissect_10_1000_direct.csv

interface's overhead. Since this overhead is not dependent on the size of the matrices, it is therefore constant and its performance impact only noticeable with small sizes. The same thing can be observed looking at the garbage collection, which also starts with a high impact that decreases steadily until not being noticeable at all, because its performance does not at all depend on matrix size.

This means that the calculations themselves only make up a small fraction of the time needed to perform the whole task. The overhead introduced by the interface accounts for over 95% of the runtime for matrix sizes of 400×400 and lower. It could thus be beneficial to develop different strategies for transferring numerical lists from Prolog into *NumPy*-arrays and back. For matrices bigger than 400×400 the complexity of matrix multiplication is no longer dominated by the overhead. For matrices with a size of 1000×1000 the calculation amounts to around 30%.

The methods of timing have the same problems of low resolution for smaller sizes due to measuring in milliseconds and measurement of real time being impacted by other programs running on the machine as described in Section 4.1.1. While there are clearly visible outliers, an underlying trend is also visible. Therefore, the outliers seem to be caused by an uneven CPU-time allotment.

4.2 Using Pre-Generated Models

In this study, I will evaluate how suited the interface is for a real world practical use case. I will examine simplicity, ease of use and speed. Furthermore, I will specifically look at the viability of using models from within Prolog that were trained and saved in Python previously.

ProB [LB03] is a model checker for the B Method [Abr96] written in SICStus Prolog and being developed at the research group *Software Engineering and Programming Languages* at the *Heinrich Heine University Düsseldorf*. ProB is capable of using different constraint solvers as its back end. These solvers all perform differently for different sets of constrains. Determining which constrain solver is capable of solving a constraint the fastest beforehand is not a trivial problem.

Dunkelau and Baldus [DB21] developed an approach to rank how suited different constraint solving back ends of ProB are to solve a specific constraint. For this, they trained decision trees and random forests. To implement their approach they used a Java API and an interface between Java and Python to train the model in Python. The trained model was then serialized and saved to disk with the Python module *Joblib*. Using the direct interface between Prolog and Python instead eliminates the need to use of Java as an intermediary.

The tool expects feature bit-vectors as its input, which are generated from the constraints. This process is not relevant for the case study and therefore omitted. Instead, I will directly use feature vectors that have been generated before. The tests will be carried out with two

Listing 6: Prothon is used to access desicion trees that were pre-trained in Python.

```
1:  :- use_module('../src/connection').
2:
3:  setup :-
4:      pyconnect,
5:      pyimport('joblib').
6:
7:  load_model(Path, Model) :-
8:      atom_codes(Path, PathStr),
9:      pycall('joblib.load', [string(PathStr)], dict([]), Model).
10:
11: predict_single_vector(Model, Features, Prediction) :-
12:     pymethod(Model, 'predict', [[Features]], dict([]), PyPrediction),
13:     pyderefall(PyPrediction, [Prediction]),
14:     pygarbage([PyPrediction]).
15:
16: test(Class, Regression) :-
17:     load_model('models/prob-tree-clf.joblib', Clf),
18:     load_model('models/prob-tree-reg.joblib', Reg),
19:     sample_feature_vector(Vector),
20:     predict_single_vector(Clf, Vector, Class),
21:     predict_single_vector(Reg, Vector, Regression).
```

decision trees – one classifier[4] and a model for regression[5] – that were trained for ProB's own back end.

The maximal time penalty of using the interface can be measured by comparing the execution times for the interface calls with the execution time of the equivalent code in Python. Since the time it takes to execute the Python functions over the Java API can not possibly be faster than executing the functions in Python directly this provides a good baseline to compare against.

I measured the time for each API call in the client individually. To compensate for fluctuations, I did the measurement 100 times and calculated the mean value out of those samples. The predictions were performed with eight different feature vectors for which I also calculated a combined mean.

The runtimes on the server were measured similarly by timing each function call individually. The means were also calculated in exactly the same way.

Listing 6 shows the entire code that is necessary to use the models from within Prolog. Using the server only requires a brief setup phase consisting of connecting to the server and

[4] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/lars-leyendecker/-/blob/1c4d15508293db74127d3904fce11d214b2e1466/prothon/server/models/prob-tree-clf.joblib

[5] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/lars-leyendecker/-/blob/1c4d15508293db74127d3904fce11d214b2e1466/prothon/server/models/prob-tree-reg.joblib

Table 1: The runtimes and overheads were measured for loading and using models with *Joblib*.

	Load Classifier	Load Regression	Predict Classifier	Pred. Regression
Python	6 ms	21 ms	0 ms	0 ms
Prothon	3 ms	21 ms	3 ms	3 ms
Overhead	−3 ms	0 ms	3 ms	3 ms

importing the necessary Python module *Joblib* into the server instance.

Loading the model is then as simple as turning the atom containing the model path into the appropriate format of a character list and making a single interface call invoking the function `joblib.load`. The path to the model needs to be specified relative to the server. The entire code required to load the model uses less than five lines of code, including the initial Prothon setup.

Getting the prediction for a feature vector on a model is achieved by an idiom common to the interface. It consists of calling a method – in this case the `predict` method of the model – and dereferencing the resulting object completely to obtain its value before finally calling the garbage collector on the intermediate object reference. Predicting a value therefore only requires three lines of Prolog code that can be wrapped in a single predicate. For all models providing the `predict` method the same predicate can be used.

The resulting times in Table 1 show that the time penalty for using the interface in this use case does not exceed 3 ms on average for any single call. Surprisingly, the overhead for loading the classifier is negative. The reason for that is an outlier value for the first sample. Performing the benchmark multiple times always reproduces this first outlier sample, which implies that it is caused by the python interpreter just starting up. A similar first outlier is also present for the interface calls, but it is much smaller there. Loading the classifier model for the first time in Python takes over 100 times longer than the mean time. However, loading it for the first time over the interface only takes around twice of the mean time. This implies that it is not an issue of caching. Otherwise, the first interface call should be similarly slow.

In conclusion, the Prothon interface allows loading modules trained in Python and using them for predictions with no more than ten lines of code and an overhead of around 3ms per prediction.

5 Related Work

Several other libraries or tools exist that provide an interface with a SICStus Prolog process or the SICStus Prolog runtime kernel for different other languages. These vary in

directionality and functionality.

The library Jasper [CWA+22, Chapter 10.20] provides an interface to Java, which can be used bidirectional, as the direction depends on which language hosts the parent application. The communication between the systems is established by either loading the SICStus runtime kernel into the Java Virtual Machine (JVM) or by loading the Java engine as a foreign resource into the Prolog system.

SICStus Prolog also supports being run as a sub-process to any language by using the JavaScript Object Notation (JSON) [CWA+22, Chapter 10.22]. This interface is completely independent of the host language and as such provides a way to use Prolog from within Python. It enables a client to initiate a sub-process of SICStus Prolog and run any number of requests to the Prolog engine. These requests can for example be a call to a predicate or a query containing multiple predicates. Backtracking is supported and the host-process can inquire multiple solutions for any given query until it eventually fails.

The model checker ProB [LB03], the core of which is implemented in SICStus Prolog, provides a Java API [KBD+20]. This interface allows communication with ProB from within Java. It is similar to the Prothon interface in multiple aspects. It uses sockets to establish a channel of communication. This allows using the output of earlier calls for influencing future calls. Another similarity is the usage of a whitelist which specifies the commands that can be used with the interface. However, every command on the whitelist is explicitly implemented in the interface. This is different to the design of Prothon which is completely agnostic towards the functions and methods that can be executed on the server.

6 Future Work

Some possible improvements to the interface can be conceptualized easily, but their implementation would exceed the scope of this thesis. In the following, I will outline those improvements briefly.

6.1 Port to Other Versions

The currently small code-base makes a port into a non-commercial and open source Prolog implementation feasible. There might be a benefit in maintaining two versions of the client so that it can be used either with the fast but commercial SICStus Prolog or with a more accessible and thus more widely used implementation such as SWI-Prolog [WSTL12].

6.2 Garbage Collection

Manual garbage collection – as described in Section 3.6 – has a number of drawbacks that go beyond convenience.

One drawback can be encountered when implementing a Prolog module that wraps the interface for a specific use case. It might be desirable to allow the user to be completely unaware of the module's implementation. However, any reference passed to the user would need to be manually handed to the garbage collector. Therefore, Python object references can not get exposed to the module's outside at any point. If the garbage collection was automatic allowing references outside the module would not pose a problem.

Another problem is that manual garbage collection does not work well with backtracking. Object creation is a side effect of most calls to the interface. If the Prolog engine reverts to a choice point before the object was created, it will still exist on the server. However, no reference to the object exists in the client. As a result the memory this object occupies on the server is lost indefinitely.

A similar problem arises if the Prolog engine backtracks to any point before a reference was handed to the garbage collector. Any attempt to use the object reference would result in an error, due to the object not being held in the server's reference dictionary anymore. Thus, it is impossible to access the object through the interface.

Additionally, manual memory management is a known source of programming errors in the form of memory leaks, which in the worst case can significantly slow down a program or even cause it to crash after nearly all the system's memory was consumed.

Automatic garbage collection should eliminate memory leaks and ease the backtracking problems significantly. It might allow for a more seamless use of the interface if used in conjunction with only immutable objects and side effect free functions and methods on the server side.

This could possibly be achieved by using the C interface of SICStus Prolog to periodically check which of the atoms referencing a Python object are still in use and sending a query to the server to delete everything except for those objects. The `keep` command already implemented in the server can be used for this purpose.

6.3 Security

In its current state the server is only suited to run locally due to severe security concerns regarding possible execution of arbitrary code. However, a remote connection to the server could provide some advantages such as running it on a much faster machine. It would thus be beneficial to implement adequate security measures allowing to connect the server to the Internet confidently. Once access to the server is granted, arbitrary code can be executed.

Therefore, requiring authorization to connect is an effective security measure.

6.4 Persistent Sessions

Having an authorization process in place would also open the possibility of persistent sessions. Currently, the connection is required to stay active throughout the entire session. However, sessions can be dropped and picked up again at will, if it is possible to match a user to their running session. This would have two benefits. First, it would be convenient to perform a time intensive task, drop the connection and reconnect later to fetch the results. Second, it would provide resilience against network failures which currently lead to the whole session being discarded.

6.5 Dereferentiation Interface

The server's parser can dereference only a discrete set of object types. Additional object types can be dereferenced in two different ways. First, a custom function can be called over the interface that returns a representation of the object which only contains types already supported by the parser. This representation can then be dereferenced using the interface. Second, the parser can be extended to be able to handle the object type directly. This eliminates the need to call an additional function when dereferencing the object.

However, using a Python interface could be a more refined approach to extending the parser. Such an interface could be realized by requiring the implementation of a single method in the classes that adhere to it. This method would simply return a string representation of the object that can be parsed by the Prothon client. This utilization of Python's object-oriented programming paradigm allows the dereferentiation logic to be contained within the object itself.

7 Conclusion

The aim of this work is to implement an interface to use Python modules that provide access to state-of-the-art machine learning algorithms and methods. Instead of supporting only a small curated selection of modules, the interface provides access to any Python module. This was achieved by shifting away from a specialized interface to a general one, which allows calling arbitrary Python functions. By switching towards a general interface it was also ensured that it does not only provide access to all current modules, but also to modules that will be created in the future.

During the case studies it was shown that this approach provides flexible access to Python modules by utilizing simple calls and idioms. Furthermore, it was proven that an application of the interface in real use cases is viable. Loading a pretrained machine learning model

can be achieved by less than five lines of Prolog code, already including the brief necessary Prothon setup, by utilizing the readily available Python module *Joblib*. The code required to perform predictions using the loaded model also does not exceed five lines. Additionally, the same code can be used to load any model that was previously saved using the same Python module. Therefore, using the Prothon interface requires orders of magnitude less code than native implementations would need.

While the interface does introduce an overhead, the case studies have proven that a speedup for tasks consisting of an extended amount of numerical calculations can be achieved. The introduced overhead scales with the amount and complexity of the data being transferred. In complex calculations such as matrix multiplication the overhead is outgrown by the time consumption of the calculation when the size of the data increases. When the amount of data exceeds a threshold, the calculation consistently performs better using the interface instead of a native implementation.

Thus, the Prothon interface can not only provide access to technologies and implementations not available in Prolog, but it can also be used to improve the performance of existing code in some cases. It was established that there is additional potential for future improvement of the handling of *NumPy*-arrays. This might significantly reduce the overhead for transferring data and thus increase the ability for the interface to speed up numerical calculations further.

Appendices

A Documentation

A.1 Prothon Representation of Python Types

How different Python types are represented in Prothon when, for example, used as arguments for function calls:

Python Type	Prothon Representation(s)	Example
List	Prolog list	`[1, 2, 3]`
Tuple	Prolog list wrapped in the *tuple* functor	`tuple([1, 2, 3])`
Dictionaries	AVL-Tree	empty
	List of key(value) in the style of options, wrapped in the *dict* functor	`dict([key(value)])`
Boolean	Atoms *true* and *false*	`true`
String	List of character codes, wrapped in *string* functor	`string("example")`
Integer	Prolog integer	`42`
Float	Prolog float	`13.37`
Object	Reference-ID, wrapped in the *py* functor	`py(12985498712)`

A.2 Public Interface

Prothon provides the following public interface:

`pyconnect`
`pyconnect(+Address)`

Connects to the server. The address defaults to the localhost if omitted.

`pycall(+Function, +Args)`
`pycall(+Function, +Args, +KeywordArgs)`
`pycall(+Function, +Args, +KeywordArgs, -Answer)`

Calls the function with the name *Function*, which must be an atom. If it is not a built-in, it needs to be prefixed with the module it belongs to with its name exactly how it was imported. *Args* contains all positional arguments in a list. *KeywordArgs* can either be an AVL-tree with only Prothon compatible strings as its keys or a list of key(value) pairs.
Answer is unified with a Python-reference to the object the function returns and in case an exception occurred on the server it is propagated. If *Answer* is omitted the return value needs to be fetched manually and therefore an eventual exception is not thrown.

```
pycall('np.array', [[1, 2, 3]], dict([dtype(string("int"))]), NpA)
```

`pymethod(+Object, +Method, +Args)`
`pymethod(+Object, +Method, +Args, +KeywordArgs)`
`pymethod(+Object, +Method, +Args, +KeywordArgs, -Answer)`

Works analogous to `pycall` except that it does call a method on the provided object.

```
pymethod(NpA, 'sort', [], dict([axis(0)]), NpA)
```

`pyanswer(-Answer)`

Fetches the most recently returned object from the server and unifies its Python-reference with *Answer*. If an exception was thrown on the server while generating the object, the exception it is propagated.

`pyimport(+Module)`
`pyimport(+Module, +Alias)`

Imports a module into the server, with an optional alias. If an alias is used the module can only be referenced by that alias.

`pygarbage(+Objects)`

Removes the objects from the server.

`pykeep(+Objects)`

Removes all objects except the given ones from the server.

`pyderef(+Object, -DerefObj)`

Dereferences the object turning it into a constant representation of its value. If the object is a collection its contents will not be dereferenced. Notably dereferencing a list of integers returns a list of object references to those integers and not the integers themselves.

```
pyderef(Subplot, tuple([Fig, Ax]))
pyderef(NpA, [E1, E2, E3])
```

`pyderefall(+Object, -DerefObj)`

Dereferences the object turning it into a constant representation of its value. For collections this is done recursively.

```
pyderefall(Npa, [1, 2, 3])
```

List of Figures

1 The interface uses a client-server architecture. Answers are only sent by the server when they are explicitly fetched by the client. 15
2 A graph depicting the execution time of multiplying two $N \times N$ matrices while traversing the AST. 19
3 A graph depicting the execution time of multiplying two $N \times N$ matrices while encoding the references directly. 20
4 A graph depicting the relative performance impact of individual interface-calls when multiplying two $N \times N$ matrices. 21

List of Tables

1 The runtimes and overheads were measured for loading and using models with *Joblib*. 24

List of Algorithms

List of Listings

1 The equality of two lists is described recursively in Prolog. 3
2 Difference lists are used to append to lists in constant time. 6
3 The term expander translates grammar clauses to regular Prolog clauses. . . 6
4 The API is used to implement matrix multiplication. 18
5 The metapredicate 'time' is used to measure execution times. 18
6 Prothon is used to access desicion trees that were pre-trained in Python. . . 23

References

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[Bra12] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 4th edition, 2012.

[CM12] Mats Carlsson and Per Mildner. Sicstus prolog—the first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.

[CR96] Alain Colmerauer and Philippe Roussel. *The Birth of Prolog*, page 331–367. Association for Computing Machinery, New York, NY, USA, 1996.

[CWA+22] Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog User's Manual*, volume 4.7.1. Swedish Institute of Computer Science, Kista, Sweden, 2022.

[DB21] Jannik Dunkelau and Leo Baldus. Ranking model checking backends for automated selection via classification and regression learning. In *3rd Workshop on Artificial Intelligence and Formal Verification*, volume 2987 of *CEUR Workshop Proceedings*, pages 83–89, September 2021.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[JD21] The Joblib Developers. Joblib: running Python functions as pipeline jobs. https://joblib.readthedocs.io/en/latest/, 2021.

[KBD+20] Philipp Körner, Jens Bendisposto, Jannik Dunkelau, Sebastian Krings, and Michael Leuschel. Integrating formal specifications into applications: the prob java api. *Formal Methods in System Design*, pages 1–28, 2020.

[KK71] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3):227–260, 1971.

[KLB+22] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. Fifty years of prolog and beyond. *CoRR*, abs/2201.10816, 2022.

[LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, volume 2805, pages 855–874, Berlin, Heidelberg, September 2003. Springer.

[Lug01] George F Luger. *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education, 4th edition, 2001.

[PD22] The Python Developers. Python 3.10.6 documentation. https://docs.python.org/3.10, 2022.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[vdWCV11] Stefan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[VR+07] Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36, 2007.

[vRD10] Guido van Rossum and Fred L Drake. *The Python Language Reference*. Python Software Foundation Amsterdam, Netherlands, 2010.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.