Towards an ontology-assisted programming language

Viktor Garske
info@v-gar.de

October 10, 2020

This working paper outlines the idea of extending the well-known imperative programming paradigm with ideas of the logical programming domain. When used as a way of metaprogramming, it can accelerate development time and simplify maintenance while providing a more human-readable source code. This paper as well sketches a reference implementation called OXPL. Its C-style syntax incorporates ideas from C++, Python, Prolog, CSS and Rust.

Programming languages evolved over the last 70 years. They are the main tool for implementing algorithms powering the economy and society as they define an interface between humans and machines. Every programming language has a specific target and purpose which has a fundamental influence on the design and features. As a result, it is possible to classify programming languages in terms of their general structure and features they offer. This classification is known as programming paradigms. This paper examines the idea of combining paradigms from two worlds being ordinarily in contrast: imperative and logic programming.

1 Traditional paradigms

There are two major paradigms programming languages can be classified into: imperative and declarative programming. Their biggest distinction is the way how to generate a solution for a given problem.

1.1 Imperative programming

One of the most natural ways to describe a program is to write down a finite series of steps how to reach a desired state. A good example is a Rubik's cube for which different algorithms can be used to solve it. A classic application for algorithms in computer science is sorting. Such algorithms are used to bring lists into a desired order. One of the simplest algorithms achieving it is known as *bubble sort* and works by iterating through the list, comparing each element with its neighbor and swap, if necessary. This procedure repeats until no swap occurs in a pass anymore.

The way of describing the solution of a problem step-by-step makes it easy to enter on the one hand. On the other hand, code can quickly become very complex and messy. As an example, *quicksort* is another variant of a sorting algorithm but its imperative implementations tend to become opaque.

1.2 Object-oriented programming

Object-oriented programming can be seen as a sub-paradigm of the just described imperative programming and combines data and code in special units representing objects. The units are

called *classes* and can be instantiated to be populated after a concrete occurrence of an object which is described by the class.

With object-oriented programming, it is easy to model real-world scenarios into a computer program. Classes represent the blueprints of possible objects and can be freely instantiated. A very important aspect is communication among objects so that they can interact with each other in order to represent real-world relationships. This paradigm is expressive and allows to construct complex structure. Several characteristic ones have been crystallized out and are known as *design patterns*.

Object-oriented programming is a very common paradigm and has numerous variations. Prototype-based programming, for example, does not use classes and focuses on object cloning and inheritance. Some languages support multi-inheritance from more than one class while others do not.

In addition to it, there are programming languages separating data and code but allowing to compose behavior using *traits*.

1.3 Logic programming

Logic programming belongs to the declarative programming paradigm. In this paradigm, the developer states the logic behind an algorithm but not the control (flow). Declarative programming describes the *what* rather than the *how* and focuses on specifying the problem.

In logic programs written in *Prolog* there facts and rules or predicates. The former expresses information about the universe of discourse, their objects and relationships among each other. Rules enables logical implications to gather more knowledge about the domain without having to specify it explicitly.

This example maps family relationships:

```
1 father(bob, john).
2 father(george, bob).
3 grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Listing 1: Simple Prolog example

The first two lines state facts: Bob is the father of John and George is the father of Bob. By making use of the rule in the third line a Prolog interpreter can deduce that George is the grandfather of John. Thus, the statement `grandfather(george, john).` is true. This is possible because of the Horn clause notation of these rules:

$$\neg p \vee \neg q \vee ... \vee \neg t \vee u$$

This one can be rearranged so that the implication emerges.

$$p \wedge q \wedge ... \wedge t \Rightarrow u$$

To throw a bridge to the proposal, let us look at another Prolog example:

```
1 extends_directly(httpserver, tcpserver).
2 extends_directly(tcpserver, server).
3 extends(X, Y) :- extends_directly(X, Y).
4 extends(X, Y) :- extends_directly(X, Z), extends(Z, Y).
```

Listing 2: Class hierachy model in Prolog

We work with this inference knowledge already today when working with class hierarchy which can be generated like in the following example:

```
1 class Server:
2     pass
3
```

```
4 class TcpServer(Server):
5     pass
6
7 class HttpServer(TcpServer):
8     pass
```

Listing 3: Class hierarchy in Python

In this example it is possible to query whether some class is a subclass of another, for example directly like with `issubclass(TcpServer, Server)` or indirectly like with `issubclass(HttpServer, Server)`.

1.4 Other approaches

There are plenty of other paradigms for special areas. They can be used in general-purpose languages or specialized domain-specific languages.

Functional programming focuses on defining, applying and composing functions for generating output based on input. A special property is the lack of side effects which makes debugging easier.

Metaprogramming can be used to generate and manipulate code of programs, including the own one. It can allow rapid development if less code is needed to specify the program flow. This can be achieved using e.g. macros or templates.

Many modern programming languages use types to categorize values, variables, objects or components they process. *Generic programming* allows to define placeholder types which can be set just before initialization.

Many programming languages incorporate multiple of the just discussed paradigms. As an example, it is possible to write imperative code in Python and include Lambda functions which technically belong to the functional programming paradigm.

1.5 Conclusion

As Kowalski [5] stated in 1974 already, traditional development of software is not human-oriented. The reason is that traditional computers operate by executing machine code which works in an imperative way and comes with a lot of peculiarities like jumps, flags and interrupts.

Higher-level languages like C++ or Python encapsulate machine code but also require a developer converting the problem into a series of imperative steps which can be automated using the computer.

Developers of programming languages tried to make imperative programming as pleasant as possible. Nevertheless, the industry faces two serious challenges: accessibility and large dependency graphs. Computers control more and more critical systems every day. Many developers rely on already coded solutions to increase their own productivity. As a result of this inflationary complexity, developers depend on libraries written by third-parties. Third-party libraries have advantages but still can be abandoned, difficult to embed or even malicious.

An enhancement for computer programming would be the shift of solution finding towards the computer. Then, developers only have to formalize the problem correctly in order to generate a program solving the issue.

This is quite difficult and might have been practiced already, if it could be easily realized. However, imperative programming was pushed forward and will likely continue to stay for the next decades.

My approach is to combine traditional programming language paradigms with assisting logical features. This allows a smooth transition between well-known imperative techniques and logical inference features which can be used to reduce programming effort by deriving code from a proper problem specification given by the developer.

This combination can also be used to orchestrate components on a more abstract basis to support service architectures like microservices.

Developers will be enabled to access more information gathered from the source code and run time. Furthermore, they should be able to define their own facts and rules themselves.

2 Ontology-assisted Experimental Programming Language

The Ontology Experimental Programming Language (OXPL) is an approach to implement the ideas of a programming language with a linked *ontology*.

The term *ontology* is a very abstract one as it has a philosophical origin. Struder et al. [9] define it as a *formal, explicit specification of shared conceptualization*. The definition can be discussed further as done in [1] or [2]. The general concept plays into our hands as programs *are* explicit specifications already. In the concept of OXPL, the ontology describes all *resources* of a program and the relations among them. OXPL uses a knowledge base containing *facts* and *rules* which are explained in more detail in a moment.

The entry barrier for such a language should be low as possible as the ontology features might take some getting used to. A programming language which is assisted by ontology features should be first and foremost a traditional programming language being based on paradigms like the imperative one.

2.1 Hello world!

"Hello world" is one of the simplest programs possible with OXPL as an introduction:

```
fn main() {
    println("hello, world");
}
```

Listing 4: Hello world!

It consists of a simple imperative main method which contains a command for printing a string, followed by a newline. New users can write their code using imperative programming style but OXPL supports ontologic programming as well as we see in a moment.

The second example focuses on input and output. This is an imperative implementation:

```
fn main() {
    var summand1 = integer::from(std::io::input("Summand 1: "));
    var summand2 = integer::from(std::io::input("Summand 2: "));
    var sum = summand1 + summand2;
    println("The sum is" + string::from(sum));
}
```

Listing 5: Imperative sum application

But it is possible to define this program with the OXPL in a completely different way:

```
instance Summand1: integer {
    std::fetchesFrom <std::io::stdin>.
    <std::io::stdin> std::io::prompts "Summand 1: ".
}

instance Summand2: integer {
    std::fetchesFrom <std::io::stdin>.
    <std::io::stdin> std::io::prompts "Summand 2: ".
}

instance Sum: std::math::sum {
    std::math::sum::hasSummand <Summand1>.
    std::math::sum::hasSummand <Summand2>.
```

```
}

fn main() {
    println("The sum is " + string::from(Sum));
}
```

Listing 6: Ontological sum application

This program as well has one sole purpose: calculate the sum of two numbers entered from the standard input the operating system provides. The way of telling the programming language about the problem is completely different. There is only one real imperative code line.

The rest of the code consists of instance declarations and facts about them. *Facts* tell the interpreter more about the semantics of the resources in this program. *Resources* include e.g. classes, instances and functions. Instances are a special case of classes and act like singletons: they are initialized at some point and preserve their value during their lifetime. In this case, the lifetime matches the run time.

2.2 Facts

In the following, we will focus more on the facts. The code states facts which can be rewritten as:

```
<Summand1> isa <integer>. /* is a = instance of */
<Summand1> std::fetchesFrom <std::io::stdin>.

<Summand2> isa <integer>.
<Summand2> std::fetchesFrom <std::io::stdin>.

<Sum> isa <std::math::sum>.
<Sum> std::math::sum::hasSummand <Summand1>.
<Sum> std::math::sum::hasSummand <Summand2>.
```

Listing 7: Explicit triple facts

Triples, also known from the semnatic web, have the same structure as a typical sentence: they consist of a subject, a predicate and an object. Every sentence ends with a period. Subjects reference resources. Objects can either reference a resource or a literal like integers or strings. Predicate has a very important role: they establish a relation between subjects and objects. This relation is usually unary or binary. Our example in Listing 7 only contains binary relations.

There is a typical relation which is often present and ties in with object-oriented programming languages. The `isa` predicate indicates an instance of a class. If one class extends another, it will be associated with its parent through the `kindOf` relation. These relations cause inheritance of facts.

It might be time-consuming to write down the subject in the declaration blocks every time. As a consequence, OXPL offers a shortcut where the subject can be left out if it refers to its proximate scope.

The alert reader might miss some facts in the previous listing. They have a special behavior as their scope is limited. If we state facts about other subjects than our current scope, they will be only true from the viewpoint they are stated in. Inheritance is possible. If a class states a fact and a class method calls another function, the facts also hold for the callee during this function call. Facts are aggregated in respect to the actual call stack. The following two statements only hold for the scope specified in the respective comment.

```
<std::io::stdin> std::io::prompts "Summand 1: ". /* holds for Summand1 */
<std::io::stdin> std::io::prompts "Summand 2: ". /* holds for Summand2 */
```

Listing 8: Scoped facts

Facts can not only be specified by using triples. There is an equivalent alternative which reminds more of the predicate logic syntax. As an example, Listing 7 could be also represented by writing the following facts:

```
1 isa(Summand1, integer).
2 std::fetchesFrom(Summand1, std::io::stdin).
3
4 isa(Summand2, integer).
5 std::fetchesFrom(Summand2, std::io::stdin).
6
7 isa(Sum, std::math::sum).
8 std::math::sum::hasSummand(Sum, Summand1).
9 std::math::sum::hasSummand(Sum, Summand2).
```

Listing 9: Explicit facts in predicate logic

The advantage of this variant is the ability to specify n-ary predicates or relations.

There are three types of facts: explicit, implicit and inferred ones. Explicit facts are directly written down into the code. They are the distinctive extension to traditional imperative programming languages. Implicit facts are gathered from the source code. A good example is the `instance Summand1: integer {}` declaration which implicitly states the fact that `<Summand1> isa <integer>`. As the reader might notice, triples enable a way of literate programming up to a certain extent.

Inferred facts emerge as a consequence of implications. Rules will be specified similarly to Prolog ones and can state coherence like

$$\forall x, y, z(extends(x, z) \land extends(z, y) \Rightarrow extends(x, y))$$

There will be a default set of basic implications needed for proper behavior of the object-oriented features and will be oriented towards traditional object-oriented languages like C++.

The goal is to provide an interface for developers to define their own rules but there are some limitations, e.g. in terms of decidability of first-order logic.

OXPL uses the closed world assumption which simplifies operations and clarifies the intuition. Everything which is not explicitly stated, either as fact or a rule, is wrong. This is possible because the program sets clear boundaries. Knowledge can be added by specifying or `importing` it explicitly. The order and position does not influence the truth value. Facts are collected in a separate pass before any line of imperative code will be executed. There are two major exceptions: scopes, as already explained, have an influence on the truth value. The same goes for imperative blocks like functions where order and position *does* count. Besides from that, the scope rules hold for imperative blocks, too. If a block ends, all of its predicated facts do not hold anymore.

This programing language wants to simplify maintenance and thus it supports modularization. To make the composition of resources as modular as possible *blocks* which are preceded by a *selector* like `class Foo` or `instance Bar` can occur multiple times as long as they address their target right: if module 1 defines a `class Car`, module 2 can only add facts and methods if its selector is `class mod1::Car` (neglecting the modularization mechanisms). If a third module wants to use facts and functions of both modules, it has to `import` both. If it only imports module 2, the additional facts about the `Car` of `mod1` are being ignored.

The result of formalization and representation of knowledge about classes, relations, instances in a universe of discourse builds this ontology. That's why I call this approach an ontology-assisted programming language. It maintains a special database about the code directly available as a tool for developers.

2.3 Example use cases of the knowledge base

There are various benefits when using ontological features. Logical inference is the most significant benefit from a proper knowledge representation and can be used user-defined.

It allows components to interact better among themselves. In addition to it, facts are also be consulted by the interpreter. The following two sections cover example applications.

2.3.1 Hooks

First, it is very easy to extend the language. Special predicates can manipulate the program flow on an opt-in basis. For example, it is possible to specify hooks:

```
fn my_function() {
    println("Hello world!");
}

fn run_before() {
    println("(early) Hello world!");
}

<run_before> std::flow::runsBefore <my_function>.

fn main() {
    my_function();
}
```

Listing 10: Hook example

The output will be:

```
(early) Hello world!
Hello world!
```

Listing 11: Output of the hook example

The hooks can also be scoped making the language very expressive.

2.3.2 Permission system

Another use case emerges from the direct flexible communication between code and interpreter. It is possible to realize a permission system which controls all handover points between the code and the outer world.

Even the `println` function communicates with this outer world by writing to the `stdout` file stream. If the interpreter is configured strictly, developers have to specifically allow even this communication:

```
allows(this, std::io::print, main).

fn main() {
    println("hello, world!");
}
```

Listing 12: Permission example

The fact in the first line allows the main function to use the `std::io::print` permission. The subject is the program itself. Every permission needs to be granted from program root level. Later, inheritance is still possible for the grantees but only for the same or a subset of the granted permissions – never more.

There will be a very limited set of default rules including the one of the previous listing in order to make it easier to get started with the language. This behavior can be easily disabled as well.

2.4 A case study: an echo server

This example shows a sample implementation of a simple echo server. The server can be configured declaratively. It is possible to implement parts of it using imperative programming.

This program uses facts for the configuration of the `class` and the interpreter.

```
import std::net;
use std::net;

/* permission */
allows(this, std::net::io, main).

/* instance = Singleton class */
class EchoServer: Server {
    /* Server-specific stuff */
    usesTransportProtocol <tcp>.
    tcp::hasBufferSize 1024.
    bindsToAddress <ip::address::Any>.
    bindsToPort 2000.
    handsNewConnectionsTo <handle_echo>.

    /* methods (overrides) can follow here */
}

fn handle_echo(con: Connection) {
    con.send(con.read_all());
    con.close();
}

fn main() {
    /* facts can also occur in imperative blocks */
    runsForever.

    var server = EchoServer();
    server.start();
}
```

Listing 13: Simple echo server

This is an artificially constructed example as the facts just map to the class properties and the constructor for the initialization of the server. The interesting part is the extension of this implemented class.

If requirements change and the echo server needs to support TLS, new facts can be added to the environment.

```
import std::crypto::tls;

/* echo server from previous listing */

fn start_tls_echo_server() {
    <EchoServer> std::crypto::tls::usesTls.
    <EchoServer> std::crypto::tls::hasCertificate "cert.pem".
    <EchoServer> std::crypto::tls::hasKey "key.pem".

    var server = EchoServer();
    server.start();
}

fn main() {
```

```
15      runsForever.
16      start_tls_echo_server();
17  }
```

Listing 14: Simple echo server with TLS

The correct TLS connection wrapping can be reached using inference: the rules of the `std::crypto::tls::usesTls` relation in combination with the `std::net::Server` class can lead to a new instance of a `TlsWrapper` and the following inferred facts:

```
1  <EchoServer.constructor> isFollowedBy <TlsWrapper.constructor>.
2  <EchoServer.destructor> isPreceededBy <TlsWrapper.destructor>.
3  <EchoServer.accept> isPreceededBy <TlsWrapper.accept>.
4  <EchoServer.receive> isPreceededBy <TlsWrapper.receive>.
5  <EchoServer.send> isPreceededBy <TlsWrapper.send>.
```

Listing 15: Inferred facts

This procedure is transparent for the handling method and developer but can be extended or manipulated as well by overriding methods and implementing functions imperatively.

2.5 Implementation and prospects

I have started to develop a toolchain for ontology applications called *ontc*. The goal is to fully specify and implement the OXPL language as well as building an interpreter, debugger and analyzer. The latter component can be used during the development and assists with the ontology and inferred facts. This component can be attributed to computer-aided software engineering.

ontc is implemented in C to ensure good performance and portability on a variety of systems. One of the first milestones will be a basic interpreter taking the knowledge base into account. Later implementations will follow a hybrid approach by precompiling code using LLVM and storing it combined with the already-inferred ontology to speed up execution time.

The development period will be estimated with one to three years for the first milestone and several years for the second one. Furthermore, a standard library will be developed in parallel which can take several years for the different stages, too.

The development repository includes a simple proof of concept already and is available on GitHub under the GNU GPL license.

3 Assumptions and Limitations

Machine code of common architectures was, is and will be imperative. This development method adds an abstraction layer to make computer programming more accessible for humans in the style of *literate programming*.

The goal yet hinges on the regarding of a high quality standard library covering various application fields. The existence of such a standard library will be assumed for this paper.

An additional layer also increases complexity making troubleshooting of special cases more difficult. Thus, good software development tools are assumed to be developed in order to scoop the productivity gain OXPL promises.

Ultimately, the language itself should stabilize during the next stages to reach acceptance. This includes a familiar and consistent syntax, API interface and standard library ontology.

4 Conclusion

This paper picked up some traditional programming languages being used currently and recognized weaknesses in accessibility and productivity. It proposed a combination of well-known

paradigms and logic programming in order to accelerate software development. Code examples were given to imply the realization of this idea. The paper finished by discussing the strategy and risks.

The corresponding project is in an early stage and has an ambitious goal but might have the potential to push some parts of software development forward.

References

[1] Johannes Busse et al. "Actually, what does "ontology" mean?" In: *Journal of computing and information technology* 23.1 (2015), pp. 29–41.

[2] Nicola Guarino, Daniel Oberle, and Steffen Staab. "What Is an Ontology?" In: May 2009, pp. 1–17. DOI: 10.1007/978-3-540-92673-3_0.

[3] Pascal Hitzler. *Semantic Web Grundlagen*. German. Berlin: Springer Berlin, 2008. ISBN: 978-3-540-33993-9.

[4] D. E. Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (Jan. 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. eprint: https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf. URL: https://doi.org/10.1093/comjnl/27.2.97.

[5] Robert Kowalski. "Predicate Logic as a Programming Language". In: IFIP Congress. Information Processing. North-Holland Publishing Company, 1974, pp. 569–574. URL: http://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf (visited on 10/07/2020).

[6] Jeff Pan. *Ontology-driven software development*. Heidelberg New York: Springer, 2013. ISBN: 9783642312250.

[7] Robert Sedgewick. *Algorithms*. Upper Saddle River, NJ: Addison-Wesley, 2011. ISBN: 978-0321573513.

[8] Heiner Stuckenschmidt. *Ontologien Konzepte, Technologien und Anwendungen*. German. Berlin, Heidelberg: Springer, 2009. ISBN: 978-3-642-05403-7.

[9] Rudi Studer, V Richard Benjamins, and Dieter Fensel. "Knowledge engineering: principles and methods". In: *Data & knowledge engineering* 25.1-2 (1998), pp. 161–197.