The Ultimate Representation of C++ Semantics

Boris Sunik
Boris.Sunik@GeneralInformationTheory.com

Annotation

The starting idea for this approach is fairly simple — since the fundamental types of C++ are quite similar to the conventional C++ classes — why not represent them with the help of standard C++ constructs. Consequently applying this idea to other C++ built-ins, the complete language could be defined in the form of a primordial library containing the explicit definition of all language fundamentals. The version of C++, extended by such expression abilities, will allow compositions of distinct collections of fundamental types, control statements and implementation mechanisms.

Unfortunately, the practical attempt to compose such a description fails immediately because several characteristics of C++ built-ins are looked at as being basically inexpressible in the conceptual system of this programming language.

This work shows how minor extensions of C++ enable the explicit definition of its complete semantics and demonstrates the general specification of C++ produced in the extended C++.

1. Introduction

The idea of formal representation of a programming language is nearly as old as programming languages themselves. The initial work was carried out by Noam Chomsky [Chomsky] who proposed a containment hierarchy of classes of formal grammars and formulated the idea of a universal grammar. The next step was taken by John Backus who developed the formal notation known as BNF (Backus Normal Form). BNF was first presented in the paper describing IAL, which was later renamed ALGOL. The method, which until now remains the only practical method used to formally represent the syntax of programming languages, was formulated by Backus as follows: "There must exist a precise description of those sequences of symbols which constitute legal IAL programs… For every legal program there must be a precise description of its 'meaning', the processor transformation which it describes, if any…" [Backus, 1960].

Despite the fact that BNF failed to achieve the second part of this objective, it inspired further intensive research, which produced miscellaneous methods for formally expressing a program's meaning. The methods developed during the last five decades are traditionally grouped into three general approaches known as denotation semantics, operational semantics and axiomatic semantics [PRATT/ZELKOWITZ].

Unfortunately, none of these approaches was ever able to attain any status outside of pure theory, because of the lack of useable results. It is very difficult to imagine that a programmer studying some programming language could extract any useful knowledge from formal specifications of this language produced with the help of known methods of semantics representations.

Taking into account the longevity and the intensity of research, its inclination to the formal mathematical models and the diversity of developed solutions, the only plausible explanation of the failure is the presence of certain fundamental constraints basically hindering the effective solution in the proposed ways.

Such a culprit exists in reality. It is the traditional view of the task of language specification, which is shared by all known methods of semantics specification. The view established in computer science in its early years was described by Jean E. Sammet in her famous book on the history of programming languages in the following way: "To define a language, some language must be used for writing the definitions. This latter is called a metalanguage. It is a general term which can include any formal notation or even English itself. Metalanguage is a relative term since it is itself a language which must be defined, and that requires a metametalanguage...." [SAMMET 1969].

Oddly enough, the distinction between a language and a metalanguage is unknown outside of computer science. The English used to write English text books is the same as that used to compose other English texts and the difference consists exclusively in the discussion's subject. And according to [searchSqlServer.com] the term "meta" is commonly used in a much more unassuming way: "*Meta* is a prefix that in most information technology usages means "an underlying definition or description." Thus, *metadata* is a definition or description of data and *metalanguage* is a definition or description of language".

Or in other words, the metalanguage needed for composing the specification of (e.g.) English language is a particular subset of English expression and English phraseology best suited for producing particular descriptions, but ***not*** a special language as it is normally assumed in computer science.

This metalanguage misinterpretation actually hindered the self-descriptiveness of the programming language, because it attributed different parts of a programming language to two incompatible systems of conceptual coordinates. The first system, introduced by the data model of a programming language, is utilised on the level of the coded algorithm and operates such notions as classes, functions, instances, and similar categories. The contrasting perspective, based on notions of linguistics, mathematics, and logic, which is complemented by permanent references to often subconscious human knowledge, is applied for representation of the language's fundamentals.

The approach proposed by this work actually enables the self-descriptiveness of a programming language by:

- *redefining this programming language in its own conceptual coordinates;*
- *extending its expression means to the degree allowing the complete representation of its own syntax and semantics.*

As a result, the concepts of a programming language are applied to all language basics, including fundamental types, control statements, the algorithms of starting, executing and finishing its program as well as to the external entities located outside of computer storage.

Theoretically, this approach could be applied to any general purpose programming language, but actually there are rather few candidates because distinct programming languages are very different in their expression power and expansion abilities. In this work, the approach is demonstrated on the example of C++, which delivers the most prominent example of language extensibility and powerfulness.

The self-describing version of C++ is designated in this work as OC++ (Open C++). The word "Open" stands for the independence of this C++ version from conventional constraints characteristic of programming languages. OC++ allows compositions of distinct collections of fundamental types, control statements and implementation mechanisms.

Open C++ should not be confused with similarly named OpenC++ [OpenC++]. These are two completely different versions of C++.

The general representation of C++ semantics is presented in the last chapter.

This work is based on the contents of Section 3 of the Theory of Meaningful Information. See [GIT-LanguageTheory] for comprehensive explanation of the approach.

2. Definition of a Programming Language

The subsequent definition of a programming language actually redefines C++ in its own conceptual coordinates. It was originally introduced in [Sunik 2003] and can be formulated as follows: *A programming language is essentially the machine code of the processor implicitly introduced by the definition of this language*. Thus, the definition of the programming language Pascal presumes a certain logical processor, which runs programs composed in Pascal code; the definition of C++ presumes another logical processor that runs program composed in this language; the definition of HTML implicitly relates to networked browsers reading the HTML content; and the definition of LISP defines the innate processor executing LISP expressions etc. Such a processor is called the ***innate processor***.

The innate processor of a high level programming language is a pure logical construct that is not purposed for whatever physical implementation. Instead a source program is either directly executed by a software interpreter running on a physical low-level processor or is translated to the machine code of a physical processor with the help of a compiler.

The relationship between a programming language and its innate processor is essentially the same as the relationship between a low level machine code and a physical processor running it. For example, the low level i386 machine code is run by processors of the i386 family. Computer processors of other architectures are controlled by codes composed in their own machine languages. Accordingly, an innate processor of C++ is controlled by C++ control structures, while an innate processor of Lisp is controlled by Lisp's expressions.

The innate processors of different programming languages possess distinct architectures, execute distinct commands and run their programs in distinct ways. Differing from simple instruction types of low-level processors, innate processors of high level languages support hierarchically built instructions.

An innate processor normally includes the main processor directly executing source code of a programming language and a low-level coprocessor executing low-level (assembler) functions.

The following describes the structure and working principles of the innate processor of C++

2.1. The Innate Processor of C++

2.1.1 Internal Memory

The internal memory accessed by an innate processor of C++ is a sequence of bytes used for the allocation of the code storage containing the running program and the data storage. The latter contains the following components:

- public and static objects;
- the program's stack with the automated variables;
- data allocated with `malloc` routine and released with `free` routine;
- data containing the identification tables of the called functions and the allocated automated variables;
- identification table of program's global objects (publics);
- array of module identification tables, each describing the content of the respective module;
- work storage used for the recognizing and processing the current command.

Other memory components are internal registers like instruction pointer (IP) and others.

2.1.2 Command Set

- control structures (*if*, *while*, *for*, *do*, *switch*, *case*, *break*, *return*, *continue*, *goto*);
- declarations and definitions;
- the function call statement and the method *execute* actually executing the called function;
- operators with built-in fundamental types (as `int operator=(int op1,int op2);…`).

2.1.3 Data

A processor manipulates instances of built-in fundamental types *char*, *short*, *int*, *float* and user-defined types (*structs*). The latter are sequences of instances of user-defined and fundamental types.

Fundamental types are built-in classes with operator functions as their methods. Their general superclass is a byte sequence allocated in the work storage. Its attributes are the address and the size in bytes. Methods of fundamental types are operator functions processing type instances.

2.1.4 Code Structures

- A program in C++ is a sequence of modules (files), which are read from the external memory into the code storage at the program's start.
- A module is a sequence of declarations and definitions constituting a unique (non-typed), unnamed instance.
- Declarations introduce classes, functions and externals. The externals are elsewhere defined instances of fundamental and user-defined types.
- Definitions represent instances (global and static) of fundamental and user-defined types actually allocated in this module.
- The communication between modules is effected by means of declaration of externals, actually defined in one of the present modules.

2.1.5 Command Execution Order

Assuming a C++ program is semantically and syntactically correct, its execution includes two steps.

The initialization step starts by reading the complete program modules, storing them in the code storage and generating the identification table of public instances as well as the array of module identification tables.

After that an innate processor consequentially processes each module. For each declaration, it creates an entry in the identification table of the respective module. For each instance definition it invokes the constructor creating a new instance. For each non-static declaration, it also creates an entry in the public table.

External declarations are stored in the temporary table for subsequent processing.

After reading all modules, an innate processor walks the temporary table of external declarations and replaces their occurrences with respective declarations/definitions from the public table.

The execution step begins with the call of the function *main*.

An innate processor reads commands, recognizes their types and passes them to the respective decoders. The method *execute*, invoked by each call of a C++ function, invokes ALU functionality when operations with fundamental types have to be performed.

Each instance object is characterized by a number of attributes such as the identifier (for a named object), the type name, and the position (absolute addresses for globals, offsets for locals and struct members, etc.).

Calls to external low-level functions invoke the low-level coprocessor.

2.1.6 Implementation

An innate processor of C++ can be implemented in three possible ways, of which only the last has a practical use:

1. Creating a physical device executing C++ code;

2. A logical device emulated by a non-optimized software interpreter running on the low-level processor. It is an application composed in the machine code which completely reproduces the innate processor's logic. The input parameter of an interpreter is a program composed in this programming language.

3. With a low level processor running a low level program that was produced by a C++ compiler and a linker. A compiler is a code-inliner that, on the basis of a source program and its own built-in knowledge of the C++ interpreter, produces a degenerate version of an interpreter reduced to the execution of this single program. The compiler pre-executes the interpreter, optimizing away all operations with constant objects, and generates the optimized code of all calls.

2.1.7 A C++ Computer

A *C++ computer* is an imaginary PC controlled by an innate processor of C++. This computer is functionally equivalent to a conventional PC, because both computers will produce the same result in the same way while executing the same program..

Principle of operations and organization of a C++ computer are also similar to those of a conventional PC. It has a keyboard, mouse and monitor and is supervised by some OS. Its programs are produced by a programmer who uses some editor to input program into a computer. After preparing a program, a programmer stores it in a file, after that it can be invoked in this or another way (e.g. from the command line).

A communication process between a programmer and a C++ computer includes several operations with a program text, each accompanied by repeated recoding of text items. Direct communication between a programmer and a computer contains minimally three codes: a) items allocated in the programmer's brain; b) identifier of keyboard's key allocated in the programmer's brain used during the typing process; c) items allocated in the binary memory of a C++ innate processor.

Reverse communication between a computer and a programmer is used to convey the program already placed into the computer storage to the programmer. This communication process normally includes three codes: a) items allocated in the binary memory of a C++ innate processor; b) external items consisting of alphabetical characters placed on either computer monitor or printed on a sheet; c) items allocated in the programmer's brain.

Yet another operation is the exchange of the program's text between computer memory and the persistent external storage like a hard disk, a flash, etc. This process normally involves two coding systems, one of which is the internal representation in the binary computer memory and the second is the specific coding characteristic to a particular information carrier. For example, hard disks use codes consisting of magnetic particles allocated on their surfaces.

3. C++ Semantics in Terms of the Proposed Approach

The above view of C++ actually closes the semantic gap traditionally emerging between the representation world of a programming language and the real world, because it regards the language as the part of the real world. According to this view, the complete C++ semantics includes not only the specifications of entities directly designated by C++ (i.e. bit structures allocated in the computer memory and algorithms of their processing) but also specifications of the following logical and physical entities:

1. *A programmer* composing *a C++ program* and the process of its communication with a computer

2. *An imaginable computer* directly executing C++ source code. This logical computer is modeled by *a conventional PC* running an imaginable non-optimized interpreter executing a source program.

3. *An imaginable non-optimized interpreter*, which is a low-level software application presumably running on a physical processor and completely reproducing the innate processor's logic. While this interpreter is never used in reality, its functionality is actually embedded in the object code generators of a C++ compiler.

4. *A compiler* that uses a source file to produce an object file representing the degenerate version of an interpreter reduced to the execution of this single program. The compiler is able to do this job because of its built-in knowledge of the C++ interpreter. The compiler pre-executes the interpreter, optimizing away all operations with constant objects, and generating the optimized code of all calls.

5. *An executable program* produced by linking together several object files.

This work is restricted to the representation of a C++ computer, which is viewed as a conventional PC controlled by a low-level processor like Intel, AMD or Arm. A computer runs a plain non-optimized C++ interpreter on its top. It is supposed that this interpreter is also written in C++ and is compiled to the machine code, so its own code will not be the subject of interpretation. Furthermore, this work is concerned exclusively with the code allocated in the memory of a C++ computer, because it is the only semantically relevant code used in this communication task.

While the biggest part of algorithms and structures associated with a C++ computer can be expressed in the standard C++, two features need language extensions for their representation.

The first is C++ control statements. Traditionally, the control statements are represented in the programming language's documentation with the help of BNF or similar meta-notations. Because this model interprets them as the structures of conventional `char` strings allocated in the code segment of a C++ computer, they theoretically can be viewed and described as user-defined types. Unsuccessfully C++ specifiers `struct`, `union`, `enum` don't support sequentially processed structures of variable lengths.

The extensions, allowing representation of control statements, are defined in 5.3.

The second is the entities beyond the representation world of C++ (real world things).

C++ objects are dynamical entities whose life span cannot exceed that of a C++ application that produced them. Thus, the definition "`unsigned char array[1000]`" defines a byte array allocated in the computer's storage. The computer storage itself couldn't be designated in this way because it was created before an application start and it will continue to exist after its end as well.

Another group of external entities are algorithms performed outside of a C++ innate processor. Just consider the process of initially starting a C++ application. While a C++ application could easily start other C++ applications, it is impossible to represent the process of starting the same applications by a computer's user, because a user's functionality is not controlled by C++.

Specification of real things requires non-executable representations (narratives) missing in C++ as well as in other programming languages. The inability to produce narratives is the fundamental restriction of the programming language. Aside from pure pragmatic reasons, this is the direct result of the theory of programming, which has never demonstrated any interest in providing whatever explicit definition of executability hence disallowing explicit definition of the opposing non-executability.

A substantial part of this work is concerned with the features of narratives and their differences from commands.

4. Principles of Non-Executable Code

4.1. Purpose

The only non-executable constructs supported by C++ are declarations of external instances and functions. Differing from other C++ constructs, declarations don't cause whatever immediate activities of an innate processor and are only conditionally necessary for programming. Many languages don't use them at all. In old "C" they were used for declaring the type of an external variable and a function's return.

Nevertheless, declarations are enormously important for C++ though not as the input of a C++ innate processor. The latter hardly needs whatever high-level features and is best served by a regular set of unsophisticated instructions organized in the way of the assembler code.

The real significance of declarations reveals the implicit communication occurring between a programmer composing the source code and a programmer studying it. This non-executable communication doesn't influence the functionality of an innate processor, but it is the integral part of the process of designing a program and is the true reason why high-level programming languages are used at all.

The information content transmitted in this communication is essentially greater than that in a communication between a programmer and an innate processor and includes along with the executable also non-executable information. The differences in the information content existing between these communications could be demonstrated in the following example.

Suppose, there is a flashlight connected to an output computer's port. The port is implemented as the byte variable located at the reserved address `0x10000`. As soon as the byte located at this address gets the value 1, the signal is transmitted to the external pin connected with the output port by intermediary of the internal bus line. Activation of the external pin switches on the power source which in turn causes the flashlight to light. Sending the value 0 into the output port turns the flashlight off.

So, in order to switch the flashlight on, a programmer (let's call him John) composes the program consisting of a single assignment "`*(char*)0x10000 = 1`".

As an expert in executing built-in commands, an innate processor of C++ will carry out the assignment, being otherwise completely unaware of the task behind the latter. Generally-speaking, an innate processor has no knowledge about the flashlight, its properties, how it is activated and even the very fact that this thing is connected to it.

On the other hand, this single-operator-program will completely fail in the communication between John and another programmer studying the source code, because the fact of assignment says absolutely nothing about the task it implements. John can fix the problem with the help of two informal expression means:

- Proper naming. Calling the program `turn_flashlight_on` explains the program's purpose to everyone acquainted with English.
- Writing comments that detail the whole task.

While the information content transmitted by John to an innate processor consists of a single assignment command, the content that is transmitted to another programmer additionally includes information about the structure and functionality of the connection line, the external pin, the power source and the attached flashlight. All these hardware entities could be effectively modeled in C++ by defining classes like `powerSource`, `externalPin`, `connectionLine`, as well as the algorithms using these classes. However, such modeling would never be adequate because these classes specify not the real pins, connection lines or power sources but the binary structures allocated in the computer storage.

In summation, the main reason why the semantics of C++ cannot be expressed in the language itself is that the information content transmitted in the second communication is inexpressible in this language.

4.2. Non-Executable Events versus Executable Activities

The relationship between non-executable events and executable activities can be demonstrated using the example of the English phrase "`the postcard burns`". This phrase is pure information designating some event but without any further associations. The following four-stage expression ladder demonstrates executable activities preceding the event of the postcard burning. Each step of this ladder, with the exception of the first, is based on the functionality of its predecessor.

- The non-executable event: "the postcard burns".
- The doing event that produces the target event: "Bill burns the postcard".
- The commanding event causing the doing event: ""burn the postcard" — said Sam to Bill".
- The event of composing a program, which has to be performed in proper time: "Sam instructed Bill to burn the postcard in case of danger".

The last stage represents the process of programming, which includes the following four steps. The only purpose of preceding event(s) consists of producing the target event(s)

1) Programming a program by a *programmer*.

2) Executing program by a *program executor*. A program executor consequentially reads program code and sends decoded instructions one by one to an instruction executor.

3) An *instruction executor* carries out the obtained instruction, producing required target event in the result. An instruction executor can be a part of a program executor (executing unit in CPU) or an external executor in cases when program executor commands external objects (e.g., a foreman commanding his team by virtue of given instructions).

4) Occurrence of (an) event(s) targeted by the instruction.

The last stage represents the process of programming, which includes the following four steps shown in Figure 1.

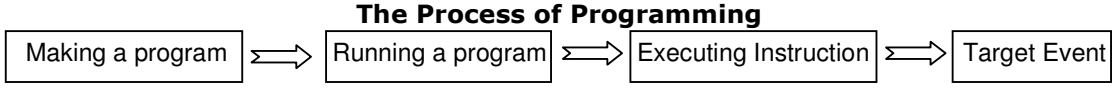

Figure 1

Since most commands in a programming language refer to functions composed in the same language, an innate processor of this language mainly instructs itself while referring other logical and physical executors only when a command refers to a function lying outside of the representation world of this programming language. In a C++ program, all C++ functions are executed by an innate processor of C++, the external Pascal functions are executed by an innate processor of Pascal and a low level function is executed by a low level physical processor.

In fact, high level innate processors are pure commanding devices doing nothing except executing these or other hierarchically organized instructions. The real target events are produced by the low level physical processors who change the states of their internal memory as a result of executing their commands. Because the storage of the modern low-level processors is built from the electronic components like capacitors, diodes and transistors, the target events of programming are essentially alterations of states of the respective electronic components.

The sequence of execution rules can be considered in the example of the command incrementing the variable defined as `"int number"`. The expression `++number` represents three commands and an event:

1. The invocation of the global operator call, whose parameters are the subroutine's address and its parameter list. The global operator call can be defined as the built-in function `operator()` whose actual parameters are the reference to the function operator ++ and the reference to a number

```
operator()(int::operator++(),number);
```

2. The global operator call invokes the specialized operator call implementing the low-level increment. The adequate designation of this operator function needs language extensions and is described in following sections. Its preliminary designation is

```
void int::operator( int::operator++()& )
```

3. The specialized operator call could be viewed as the assembler implemented function without explicit parameters. The class pointer `this` is put into the register `EAX`

```
//the function increments a double word whose address is located in the register EAX
void int:: operator( int::operator++()& )
{
    __asm inc dword ptr [EAX]
};
```

The low-level command `inc` is executed by the low-level physical processor and produces the target event of enlarging the variable `number`.

4. The target event produced by a physical processor consists of changing a four byte value located at an address set in EAX. The target event that can be designated as `int& int::operator++()` but it could not be expressed in C++ prohibiting non-executable representations.

4.3. Semantics

Imperative communications are means of control. A text reader reads a command, recognizes it and executes the requested action. The presumption of the imperative communication is the practically existing ability of a command executor to produce required actions with required objects.

To the contrary, non-executable communications are not presuming whatever activities of a text reader exceeding text processing. Narratives are sequences of data items designating miscellaneous entities (objects, events and pluralities) without further implications. A reader of a narrative text is essentially *an information collector*, which receives information about some entity without influencing the latter. Narratives are the dominating code-form of natural languages.

The semantics of the narrative text is defined by the observing activities of full-fledged information collectors. An observer recognizes external objects and events by persistently watching changes in sensory feelings. The most complex of all possible observers are human beings, whose senses include many millions of receptors, but in general it can be also generated by an arbitrary living creature or an artificial device – presuming they possess abilities for registering external events. Thus, such artificial devices as motions detectors are able to recognize changes in their immediate environment and react accordingly.

Also when the organization of the human's brain is completely different from that of digital computer, the observing functionality can be well demonstrated in terms of computer algorithms as described below.

1. Assume there is a simple application measuring the characteristics of a certain process every second and that the same process is simultaneously monitored by many mutually independent application instances. The monitoring algorithm of an application instance includes the following three steps:

a) investigating the observed process and producing its estimation in the form of two `int` numbers `int1` and `int2`;

b) comparing these numbers with the help of the following comparison function;

```
enum Result {lesser, equal, greater};
Result compare(int le,int ri)
```

c) using the result of comparison in some unspecified way.

This application can be built in two distinct ways, the simplest of which is to allow application instances to run completely independent from each other. Each instance obtains numbers separately and makes the comparison by invoking `compare`.

More sophisticated design allows communication between application instances. The first instance, which succeeded in executing the monitoring algorithm in the current second, sends results to other instances with the help of a message-sending mechanism. Other instances get the comparison result and can use it without any measurements.

The message with the comparison result represents the simplest non-executable text. It allows only three values and its semantics is completely understood by the message readers, because each of them knows how to produce the same value.

2. The next stage of language complication can be demonstrated by the application extracting more information from the monitored object. An application instance gets values of four variables allocated in the memory space of the observed process and compares them with each other. The number of needed comparisons is 6 (3 + 2 + 1) and the representation of a single comparison needs at least three elements, which could be represented by the following structure:

```
struct Message
{
    Result res;
    char   first_num;      //id from 0 till 3
    char   second_num;     //id from 0 till 3
}
```

3. The last example describes the process of registering changes, which constitutes the main functionality of an observer. A change is the alteration of an entity recognized by comparing its values.

An application watching a program run by a C++ innate processor registers changes of `int` objects occurring in the program's memory. An application instance runs in an endless loop, persistently registering allocations, deallocations and changes of `int` variables by comparing their stored and new values. Each application instance possesses the set with the addresses and the last values of monitored variables. As soon as it registers the change, it transmits this knowledge to other instances.

The process uses the variant of comparison algorithm returning values of the enum `Change` defined as

```
enum Change  {grows, unchanged, decreases};
Change  compare_with_old(int new_val,int old_val);
```

The complete non-executable language used for communicating information between application instances has to include three statements representing allocations, changes and deallocations. The sentence used for representation of changes has to include the value of `Change` and the variable's address something like:

```
struct Message
{
    Change res;
    int* adr;
}
```

4.4. Summary of Non-Executable Code

1. Representations of events (changes) are collections of descriptors produced by external observers. An observer performs miscellaneous comparisons and searches and express their results in the form of non-executable representations. Non-executable code doesn't presume whatever activities of a text reader exceeding text processing.

Imperative code consists of controls used to command a code reader. A text reader reads a command, recognizes it and executes the requested action. The presumption of the imperative communication is the ability of a command executor to produce required actions with required objects.

2. In actual fact, each imperative text is associated with two differently organized logical machines: a *text executor* and an *observer* watching the executing process. Consequentially, the semantics of this text includes at least three code collections: the text itself, the narrative text describing the executing process and the narrative text representing the results of process observation.

The number of representations can be bigger however. Thus the semantics of the C++ command `++number` includes the command itself; the assembler instructions used for implementing the specialized operator call and non-executable specifications of:

- the event or recognizing and executing the command `++number` performed by the global operator call;
- the event of recognizing and executing the specialized operator call;
- the target event of increasing number.

3. An event can be interpreted as a dynamic object without methods, whose body consists of temporarily available constants being comparison results. The operator call producing an event has to be considered as the event's constructor.

4. The communication between a programmer and an innate processor of C++ is not the only one possible in this language. Another one is the implicit communication occurring between a programmer composing the source code and a programmer studying it. This non-executable communication doesn't influence the functionality of an innate processor, but it is the integral part of the process of designing a program and is the true reason why high-level programming languages are used at all.

5. The main reason why the semantics of C++ cannot be conveyed in this language is the failure of the latter in expressing the information content transmitted in the second communication.

5. The Extensions of Open C++

5.1. Commands and Events

Commands are input parameters of respective execution handlers. An execution handler processes the command, invoking required algorithms in the results. The upper execution handler, built into a C++ innate processor, is the operator function `operator()`, which executes all user functions and methods by invoking the specialized invocation operator making exact action. Thus, the execution of the C++ command "`++number`" actually invokes the aforementioned operator, which in turn invokes the specialized function `operator()` enlarging `number` by 1. The following extensions allow the adequate representation of such algorithms:

1. *The function can return an occurrence of another function*. The function `call` defined below produces an occurrence of either `fone()` or `ftwo()`.

```
enum Command{one,two};
void fone();
void ftwo();
//functions is array of references to fone, ftwo
( void(&)() ) functions[Command::two] = {fone,ftwo};
( functions[command]() ) call(const Command command);
//execution of commands one, two
call(one);          // invokes fone()
call(two);          // invokes ftwo()
```

2. *The function return can be used for defining function overload*. Consider the following C++ code:

```
int   func(int);
float func(int);
func(3);            //error, both overload are differed only by their return parameter
```

The call of `func` causes an error message because two overloads differ from each other by the return parameter only. To resolve the issue, the output parameter can be additionally defined in the list of input parameters with the help of the specifier `out`. So OC++ allows the code to be written as:

```
int   func(int);
float func(int, out float);
//or yet shorter
float func(int,out);
...
func(3);            //invokes int func(int)
func(3,float);      //invokes float func(int)
```

This extension actually enables the standard declaration syntax for functions

```
cfunction func(int, out float);          // cfunction is the standard C function
```

3. *The specifier ^^ is used for designation of non-executable events*. So the process of executing a command is designated as `++number` while `^^++number` designates a non-executable event of enlarging of `number`.

All that allows the following representation of operator functions and a non-executable event described in 4.2.

```
(cfunction()& ) operator()(…);                  //the main operator call can take any parameters
int& ^^int::operator++();                       //non-executable event of enlarging of int object
(^^int::operator++())int::operator()(out);//the specialized operator call producing event of enlarging
```

5.2. Non-Executable Representations

While the specifier "^^" could be used for the designation of non-executable activities, it alone is not sufficient for the effective representation of narratives requiring representations of structures like those described in 4.3. The obstruction is the C++ misconception implying that condition identifiers like “<”, “==” and similar stand for names of operator functions. Particularly, this interpretation of condition is used for building user-defined operator functions like

```
bool operator<  (SomeType &li, SomeType &ri);
bool operator== (SomeType &li, SomeType &ri);
```

In reality, condition names are output values returned by respective comparison functions.

The purpose of output values is basically the same as the purpose of input values and consists of unambiguous specification of occurrences of algorithm's branches, but in C++ only input parameters have this role. The output parameters are considered as rvalues that could only be used in conditions and assignments. This usage follows from the design of a programming language proposed for making questions like “is num1 zero?” but not for making affirmations like “num1 is zero!”.

Let's consider the problem in the example of the function `compare` that assesses the relationship existing between input objects and returns an enum member characterizing the result of the comparison.

```
enum Result {lesser, equal, greater};
Result compare(int le,int ri)
{
    if( le <  ri ) return lesser;
    if( le == ri ) return equal;
    return greater;
};
```

The function `compare` could be viewed as a generic algorithm, allowing three distinct specializations defined by the combinations of actual parameters (in OC++, values `lesser`, `equal`, `compare` are considered as subtypes of `Result`). An instance of specialization is an event produced by the respective operator call. This allows the following definitions of event types:

```
lesser  ^^compare(out lesser,  int le, int ri){le < ri;           return lesser;}
equal   ^^compare(out equal,   int le, int ri){le < ri; le == ri; return equal;}
greater ^^compare(out greater, int le, int ri){le < ri; le == ri; return greater;}
```

An instance of the algorithm's specialization is essentially its value. The function `compare` allows three possible values.

- `^^compare(lesser,a,b)` if execution of `compare(a,b)` produces `lesser`
- `^^compare(equal,a,b)` if execution of `compare(a,b)` produces `equal`
- `^^compare(greater,a,b)` if execution of `compare(a,b)` produces `greater`

Assuming that enum constants should only be used as return parameters of `compare` the references can be simplified in the following way:

- `lesser(a,b)` if occurrence of `^^compare(a,b)` produces `lesser`
- `equal(a,b)` if occurrence of `^^compare(a,b)` produces `equal`
- `greater(a,b)` if occurrence of `^^compare(a,b)` produces `greater`

A value like `lesser(a,b)` could also be examined by the proofing function, which checks whether the relationship between `a` and `b` is really equal to this value and returns the constants `true` or `false` in the results.

The following OC++ features support references to the algorithm branches.

1. The set of possible branches of an algorithm can be defined with the help of info statements. The declaration of an info statement consists of the keyword `statement` following the declaration of the comparison function. The info statement designating results of the function `compare` is as follows

```
statement Result compare(int p1,int p2);
lesser(num,3);                          //means num is lesser than 3
```

2. *Identifiers can be composed from two different character sets.* The conventional words consist of letters, digits, and the sign “_”. The special words are formed from characters traditionally utilized for operators (“+”, “-“, “*”, “/”).

Both kinds of words possess the same rights. The name may include characters taken from only one alphabet. Hence, names composed from different alphabets may be distinguished without additional separators. In the expression “a=b+c” , object names (a, b, c) are automatically separated from function names (“=”, “+”).

```
proc ---(int);                          //this is the declaration of the procedure with the name “---“
```

This allows statements "<", ">", "==" to be defined as follows:

```
enum Result {<,>,==};
statement  Result operator compare(int a,int b);
```

3. *Any function with one or two input parameters can be declared as binary or unary operator*

```
void operator foo(int k);
k foo;
```

Differences between the unary prefix and postfix operator functions can be expressed by declaring the postfix operator functions with two parameters, first of which is empty. The following example shows the differences between C++ and OC++ in the declaration of the postfix operator++

```
Object operator++(Object &obj, int); // obj++ the notation of C++
Object operator++(,Object &obj);     // obj++ the notation of OC++
```

The operator precedence of operator functions could be defined in the extended attribute syntax like

```
__priority(+12) void operator foo(); // left to right (+) on the twelve's place in the precedence table
```

4. *C++ conditions are represented with the help of the built-in* `operator condition`*.* This operator function gets an info statement and checks its trustworthiness by executing the comparison function. It returns the `true` or `false` in the results of the comparison between the given and the actual comparison returns.

```
if (a lesser b)…;                       //actually means if(operator condition(a lesser b))
a lesser b                              //correspond to operator condition( a lesser b );
lesser(a,b)                             //correspond to operator condition( lesser(a,b) );
```

A pure assertion without `operator condition` can be defined in OC++ with the help of the specifier ^^ before the output parameter value:

```
a ^^lesser b                            //pure  assertion data
^^lesser(a,b)                           //alternative form of pure assertion
```

5.3. Representaton of the Control Structures

Consider the definition

```
char* mystring = "mytext";
```

The difference between C++ and OC++ consists of the number of strings, which can be identified in this code. According to C++ the only string there is "`mytext`" assigned to the char pointer `mystring`. In OC++, the code itself is viewed as the hierarchically organized text. allocated in the code segment of an innate processor. The strings in the text are quite similar to conventional string literals, but differ from the latter in that they don't possess finishing zeroes.

The structure of the code items of C++ could be effectively represented in terms of structures, unions, arrays and enumerations, which however is not possible without upgrading respective C++ specifiers. The main problem is that C++ structures consist of directly accessed components of fixed lengths while code structures have consequentially accessed components of variables lengths.

Consider the restrictions of C++ in the example of identifiers, which according to the language standard, are sequences of letters, digits and underscores, starting with a letter or an underscore. C++ allows for such a structure to be represented in roughly the following way:

```
union letter_underscore;                //letter or underscore
union letter_underscore_digit;          //letter, digit or underscore

template<int length> struct identifier
{
    letter_underscore       begin;
    letter_underscore_digit rest[length];
}
```

Despite the fact that this code looks like a real representation of identifiers, it has the four following deficiencies:

1) Because the class `identifier` is a template type with `length` parameter, the constants `ident` and `ident1` are interpreted as instances of distinct template types `identifier<5>` and `identifier<6>`. Differing from that, the common sense understanding interprets them as the instances of same type `identifier` without any regard to their lengths.

2) The field `rest` is optional and may be missing, but C++ disallows optional components.

3) Components of C++ structures can be directly accessed using their names or offsets from the structure's start, whereas non-beginning components of lexical items (in this case `rest`) can only be accessed as a result of the consequential scan (in the name "val" `rest` consists of "al", but it is missing in the name "v"). A representation of lexical structures has to guarantee the consecutive access to its elements.

4) Digits and letters are restrictions of the type `char`, allowing some values of `char` type while disallowing others. Restrictions are not supported by C++.

The following OC++ extensions solve the aforementioned problems.

1. OC++ types can be objects of variable lengths. The class `identifier` can be represented as:

```
struct identifier
{
    letter_underscore begin;
    letter_underscore_digit rest[];              //any length of this array is allowed
}
```

2. Optional components can be represented as unions with a `void` element.

```
struct identifier
{
    letter_underscore begin;
    union
    {
        void v;
        letter_underscore_digit rest[];
    }
}
```

3. Definition of unnamed components automatically excludes them from direct referencing:

```
struct identifier
{
    letter_underscore;
    union
    {
        void;
        letter_underscore_digit [];
    }
}
```

4. A restriction of a type is defined as an enumeration of unnamed values of this type.

```
enum digit :char{'0','1', '2', '3', '4', '5', '6', '7', '8', '9'};
digit('0') //digit value
char('0')  //char value
'0'        //either char or digit
enum letter :char{'a','b',...};
enum underscore :char{'_'};
```

5. Unions are superclasses of all their components. The union `letter_underscore_digit` is the superclass of classes `letter`, `underscore`, `digit`

```
union letter_underscore_digit
{
    letter;
    underscore;
    digit;
}
void foo(letter_underscore_digit);
```

```
foo('3')    //correct digit derives from letter_underscore_digit
```

The following definition introduces the upper level `operator()` with its input parameters. The functions/methods invoked by this operator function are code objects of the type `cfunction` whose structure consists of formal parameters and the block statement. That is to say, the definition like `int foo(int p){...}` in reality introduces the object `cfunction foo(input_parameters("int p")output_parameter("int"),body("{...}") )`. Accordingly the call `foo(a)` actually invokes `cfunction::operator(parameter_list)`.

In order to improve readability, the declaration order is reversed relative to original C++. The latter requires that the definition of an included class has to appear before the definition of the enclosing one.

```
typedef char(" ") blank;
union blanks{void; blank [];}                    //optional blanks
void cfunction::operator()(parameter_list;);
struct parameter_list
{
    char = "(";
    blanks;
    parameter;
    par_sublist;
    char = ")";
};
union parameter {expression; identifier; literal;};
//for the sake of space the structures expression and literal are not described here
union par_sublist
{
    void;
    struct
    {
        char =",";
        blanks;
        parameter;
        blanks;
        par_sublist;
    }
};
```

6. General Semantics of C++

The syntax of C++ is defined by the structure of the code items stored in the code segment of a C++ computer. The following fragmentary declarations show the syntactical structures of C++:

```
union module_level_declaration
{
    function_level_declaration;
    function_definition;                         //the definition of a function or a method
}

union function_level_declaration
{
    class_definition;                            //struct of class definition
    instance_declaration;                        //struct of instance declaration
    function_declaration;                        //struct of function prototype
    instance_definiton;
}

//a module consists of the set of declarations and definitions.
//The object Cpp_module is produced by a programmer
//and is used by Cpp_computer.execute_program()
typedef module_level_declaration cpp_module[];
```

```
struct code_block
{                                                   //the code block is the sequence of statements
    ”{”;                                            //enclosed in braces
    struct
    {
        blanks;
        statement;
        blanks;
        “;”;
    }[];
    blanks;
    “}”;
}
typedef identifier sid;                             //identifier of the control function
//control statements
union control_statement                             //statements in a block
{
    function_level_declaration;
    struct{sid(“if”);blanks;code_if;};              //code_if  is the structure of the if statement
    struct{sid(“switch”);blanks;code_switch;};//other statement structures are code_switch,
    struct{sid(“for”);blanks;code_for;);            //code_for, code_while, code_do
    struct{sid(“while”);blanks;code_while;);        //code_identifier, code_expression
    struct{sid(“do”),blanks;code_do;};
    struct{sid(“goto”);blanks;code_identifier;};
    struct{sid(“break”);};
    struct{sid(“continue”);};
    struct{sid(“return”);blanks;code_expression;);
    code_block;                                     // block statement
    struct code_expression{...};                    // a C++ expression
    struct code_call;                               //a  call statement
    void operator();                                //the invocation method
}

union code_call
{
    struct code_function_call{...};                 //function call statement
    struct code_operator_call{...};                 //binary and unary operators are omitted here
    virtual void operator() = 0;
}

union function_declaration
{
    operator_prototype;
    function_prototype;
}

struct cfunction_definition                         //definition of a C-function or a method
{
    function_prototype;
    code_block;
}

struct afunction_definition                         //definition of an assembler function
{
    function_prototype;
    __asm code_block;                               //the assembler code is designated as the
}                                                   // assembler code_block

union function_definition
{
    cfunction_definition;
    afunction_definition;
}
```

The interpreter is the only public method of `Cpp_computer` and is invoked from the command line by a `user`. The `operator()` that invokes the interpreter is defined outside the C++ representation world and can therefore be declared as non-executable code.

```
Cpp_computer::execute_program() ^^user::operator()(...);
```

The process of executing a program involves the following routines:

• The procedure `^^user::operator()` starts the method `Cpp_computer::execute_program` that initializes the program and invokes `Cpp_computer::cfunction::operator()` with the function `main`.

• The method `Cpp_computer::cfunction::operator()` invokes `Cpp_computer::execute_cfunc`, which consequentially reads the control statements of the function `main` and invokes respective controlling procedures etc.

• The controlling procedures process control statements `if`, `for`, `switch`, `do`, `while`, `goto`, `continue`, `break`, `return`, `block_executor`, call operators, declarators and expressions.

It is assumed that all C++ intrinsics including the method of fundamental types and the routines processing control statements are assembler functions.

The following fragmentary declarations show the overall structure of `Cpp_computer`. The innate processor in this specification is the method `Cpp_computer::execute_program`.

```
//the command line procedure starting the program
Cpp_computer::execute_program() ^^user::operator()(...);

class Cpp_computer
{
public:
    //the method executing the program
    void ^^execute_program (char**filelist,int filenum, char**arglist, int argnum);

private:
    struct afunction;
    struct cfunction;

    union function
    {
        cfunction;
        afunction;
    }

    enum invoked :afunction
    {
        void execute_asm  (afunction* ac = IP);     //the method running assembler code
        void execute_cfunc (cfunction* fc = IP);    //the method running the function
        enum basic_function :afunction;             //operator functions of fundamental classes
    }

    enum controlling  :afunction                    // routines processing control statements
    controlling control_statement::operator()(...);//operator call invoking controlling routines

    ....
    //fundamental types
    class int{...};
    class short{...};
    class char{...};
    class float{...};
    class bool{...};
    class unsigned_int{...};
    class unsigned_short{...};
    class unsigned_char{...};
    ....
```

```
    unsigned char memory[];                    //internal memory of an innate processor
    unsigned char* IP;                         //instruction pointer
    char *code_segment;                        //memory with the source code of a C++ program
    char *data_segment;                        //memory with the data of a C++ program
};
```

The following specification details the types `controlling`, `cfunction`, `afunction`, and `basic_functions`

```
class cfunction
{
    cfunction_definition body;
    cfunction(cfunction_definition);
public:
    void operator(parameter_list);             //the call of a C-function or a C++ method
}

class afunction
{
    afunction_definition body;
    afunction(afunction_definition);
public:
    void operator(parameter_list);             //the call of an assembler function
}

enum controlling :afunction                    //processors of respective control statements
{
    void if(code_if* = IP);
    void switch(code_switch* code = IP);
    void for(code_for* code = IP);
    void while(code_while* code = IP);
    void do(code_do* code = IP };
    void goto(code_identifier* code = IP);
    void break();
    void continue();
    void return{code_expression* code = IP);
    void block_processor(code_block* block = IP);
    void declaration_processor(function_level_declaration* code = IP);
    invoked[] expression_processor(code_expression* code = IP);
    invoked operator()(code_call* code = IP);
}

enum basic_function :afunction
{
    int   int::operator+(int);
    int   int::operator-(int);
    int   int::operator*(int);
    ....
    short short::operator+(short);
    short short::operator-(short);
    short short::operator*(short);
    ....
    char  char::operator+(char);
    char  char::operator-(char);
    char  char::operator*(char);
    ....
    float float::operator+(float);
    float float::operator-(float);
    float float::operator*(float);
    ....
}

struct statement_value;                        //a value of a statement function
enum bool{false,true};
```

```
bool operator condition(statement_value);
//statement used by operator condition
enum eBEL {<,>,==};
enum eENe {==, !=};
enum eGe{>=,NOTGe};
template<class T>statement eBEL compare(out, T le, T ri);
template<class T>statement eENe compare(out, T le, T ri);
template<class T>statement eGe  compare(out, T le, T ri);
...
statement<int> eBEL compare;                    //specialization of statement template
statement<int> eENe compare;                    //specialization of statement template
statement<int> eGe  compare;                    //specialization of statement template
...
statement<float> eBEL compare;                  //specialization of statement template
statement<float> eENe compare;                  //specialization of statement template
statement<float> eGe  compare;                  //specialization of statement template
...
```

7. Conclusions

1. The definition given in this work actually redefines a programming language in its own conceptual coordinates. As a consequence, a programming language becomes self-descriptive and can be used for the specification of its own semantics provided it is extended by facilities to compose non-executable narrative descriptions.

2. The proposed extensions of C++ can be used for producing the complete specification of its own semantics including all logical and real entities related to the functionality of this language. Differing from the standard C++, the extended language version allows miscellaneous implementations, which now requires the creation of completely new languages.

3. Non-executable code is the actual missing link between the closed representation world of C++ and the real world. The self-descriptiveness is not the only feature enabled by the proposed extensions. Another one is the ability to compose C++ projects combining the conventional executable code with C++ specifications of associated external entities, like devices controlled by the executable code or the purposes of a program's users. The functionality, which can be implemented by such projects, is currently reserved for multi-language systems including, among others, specification languages and modeling tools like UML.

4. The ultimate result of the demonstrated extensions is a full-scale non-executable language, which can be used for the representation of all information that is expressed in any existing language whether natural or artificial. The short description of such a language can be found at [GIT-T].

8. Bibliography

1. BACKUS W. 1959: The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, ICIP Proceedings, Paris, p. 125-132

2. Ellis M. and Stroustrup B. The Annotated C++ Reference Manual. Reading, MA: Addison-Wesley, 1990.

3. CHOMSKY Noam 1956: "Three models for the description of language", in: IRE Transactions on Information Theory 2, p. 113–124

4. CHOMSKY Noam 1957: Syntactic Structures, The Hague: Mouton

5. GIT-LanguageTheory http://generalinformationtheory.com/content3.php

6. GIT-T http://generalinformationtheory.com/content4.php

7. OpenC++ http://opencxx.sourceforge.net/

8. PRATT Terrence/ZELKOWITZ Marvin 2000: Programming Languages: Design and Implementation (4th Edition), Prentice Hall

9. Searchsqlserver.com http://searchsqlserver.techtarget.com/definition/meta

10. Sunik Boris. The paradigm of Open C++, SIGPLAN Notices, June 2003.

11. UML http://www.uml.org/