Acta Cybernetica **22** (2015) 5–20.

Automatic Checking of the Usage of the C++11 Move Semantics

Áron Baráth* and Zoltán Porkoláb*

Abstract

The C++ programming language is a favorable choice when implementing high performance applications, like real-time and embedded programming, large telecommunication systems, financial simulations, as well as a wide range of other speed sensitive programs. While C++ has all the facilities to handle the computer hardware without compromises, the copy based value semantics of assignment is a common source of performance degradation. New language features, like the move semantics were introduced recently to serve an instrument to avoid unnecessary copies. Unfortunately, correct usage of move semantics is not trivial, and unintentional expensive copies of C++ objects – like copying containers instead of using move semantics – may determine the main (worst-case) time characteristics of the programs. In this paper we introduce a new approach of investigating performance bottlenecks for C++ programs, which operates at language source level and targets the move semantics of the C++ programming language. We detect copies occurring in operations marked as move operations, i.e. intended not containing expensive copy actions. Move operations are marked with generalized attributes – a new language feature introduced to C++11 standard. We implemented a tool prototype to detect such copy/move semantic errors in C++ programs. Our prototype is using the open source LLVM/Clang parser infrastructure, therefore highly portable.

Keywords: C++ runtime performance, C++ generalized attributes, C++ move semantics, static analysis

1 Introduction

In this paper we introduce a new approach of automated checking the correct usage of the move semantics introduced by the C++11 standard [15]. As copying objects instead of moving them is considered one of the major performance bottlenecks in C++, the proper usage of the move semantics may determine the runtime performance for C++ programs.

*Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary, E-mail: {baratharon,gsd}@caesar.elte.hu

DOI: 10.14232/actacyb.22.1.2015.2

The runtime performance can be estimated in various ways, depending on which program features should be calculated. The most obvious, but also the most difficult approach is the quantitative *time* estimation (e.g. in seconds) [21, 8]. Such analysis is a complex and difficult area, because of the processor's cache operations, interrupt routines and the operating system impact the execution time. Also, it is very difficult to replicate a specified scenario. Estimating or measuring the execution time as an absolute quantity therefore is still an unsolved problem for the most popular programming languages, like C++.

In the same time, performance estimation is also important on the source code level. When evaluating a library which can be compiled on different platforms with different compilers we are interested not in absolute or approximated time but rather the number of certain elementary statements, e.g. number of (expensive) copy instructions of certain types. This approach allows us to process the code at the language source level, which is architecture independent. At the language level the number of assignments, comparisons, and copies can be measured. These discrete values can be capped by a predefined limit and can be checked by static analysis tools. Also, using such tools serious programming errors can be detected, what none of the low-level estimations can reveal.

In this paper, we introduce a method to detect move semantics errors in C++ programs. The method is based on the explicit labeling of those operations that the designer intended as *move operations*, i.e. not to include expensive copy operations. The operations excluded can be copy constructor or assignment operator calls as well as parameter passing or returning by value of complex types. Inside move operations we allow only the call of other move operations and the manipulation (copy) of non-aggregated built in types.

In order to test our method we implemented a prototype tool based on the public domain LLVM/Clang C++ compiler frontend [9, 11]. LLVM/Clang is a highly portable emerging compiler infrastructure planned explicitly as a reusable object-oriented library. Using Clang we analyze the abstract syntax tree of the source and recognize the mistakes committed in move operations and report them to the user.

This paper is organized as follows: in Section 2 we discuss the necessity of the move semantics in C++, and we present use cases for the move semantics and show possible mistakes related with. In Section 3 we present a new C++11 feature, called generalized attributes, and we use this mechanism to annotate the source code with expected semantics information. Also, we introduce our Clang-based tool to check semantics and display error messages to the mistakes. We evaluate our method and the prototype tool in Section 4. In Section 5 we briefly present *Welltpye*, our prototype language, in which semantic information can be marked explicitly. We also discuss our future plans to extend our C++ move semantics checker. The paper concludes in Section 6.

2 C++ move semantics

Copy semantics play a central role in modern object-oriented languages. Certain languages have reference semantics. Objects are created in a (possibly managed) heap, and are referred by unique handlers (pointers, references). When we apply assignment between such objects the actual operation has effect only on the handlers: the handler of the left side of the assignment will refer the object referred by the right side. No data is copied between the referred objects. This strategy is called as shallow copy. For deep copy programmers have to define specific user operations, like the `clone` function in Java [5].

The C and C++ languages have value semantics [7, 16]. Objects of C and C++ – either having built-in or user defined types – can exist in the stack, in the static memory or in the heap. In all cases variables identify the raw set of bits of objects without immediate handlers. When we apply assignment, we copy raw bytes by default.

Object-oriented languages are designed to give programmers the possibility of creating new types in the form of classes. Such types may be constructed from other complex types in a recursive manner and copying their raw bytes may not be the proper copy semantics. For such cases programmers in C++ may define copy constructor for initialization and the `operator=` for assignment. These operations usually implemented using the already defined copy operations of subobjects or the explicit copy instructions decided by the programmer. When no user defined copy constructor or assignment operator have been provided the default memberwise copy operations will be applied.

While this behavior is very convenient when we want to encapsulate the implementation of classes and building higher abstractions, it may also be a cause of serious performance issues. In case of complex classes, like matrices, vectors, lists, a single assignment may cascading down to a huge number of byte-level copies. This phenomenon exists for the containers of the C++ standard library too. Temporary objects created during the evaluation of expressions are also critical performance bottlenecks. The used `new` and `delete` operators, the overhead of the extra loops and memory access are costly operations.

Todd Veldhuizen investigated this issue, and suggested C++ expression templates [19] and template metaprogramming [2, 6, 14, 20] as a solution. The basic idea is to avoid the creation of temporaries, and unnecessary copies, but "steal" the resources of the operands, i.e. move the ownership of the data representation between assigned objects. Such *move* operations can be implemented library-based, like the `Boost.Move` library [1] with overloading the original *copy* operations. Library-based solutions, however, lack to distinguish objects which are destroyable, i.e. their resources can be safely moved out.

To distinguish non-destroyable objects from possible sources of move operations, i.e. those which can be destroyed language support is required. The C++11 standard has introduced a new reference type: the *rvalue* reference [15, 17]. In the source code, the rvalue references have a new syntax `&&` to yield reference to destroyable objects. Using this syntax, constructors, copy constructors and as-

```
class Base
{
public:
  Base(const Base& rhs);            // copy ctor
  Base(Base&& rhs);                 // move ctor
  Base& operator=(const Base& rhs); // copy assignment
  Base& operator=(Base&& rhs);      // move assignment

// ...
};
```

Figure 1: Copy- and move constructors.

```
Base f();

void g(Base&& arg_)
{
  Base var1 = arg_;   // applies copy:  Base::Base(const Base&)
  Base var2 = f();    // applies move:  Base::Base(Base&&)
}                     // arg_ goes out of scope here
```

Figure 2: Named rvalue explained.

signment operators can be overloaded with multiple types. Constructors and assignment operators with rvalue parameters are called *move constructor* and *move assignment*, as we can see in Figure 1. The move constructor and move assignment changes the ownership of the data representation of the argument object, and set it to an empty but valid (destroyable) state.

Note that, the actual parameter passed in the place of an overloaded constructor may be either an rvalue or an lvalue. When the object passed as parameter is referred by a name, then it is lvalue, otherwise it may be an rvalue. This rule – often referred as "if it has a name" rule – is explained in Figure 2.

Unfortunately, this is one of the situations where C++ programmers can easily make mistakes, and no errors or warnings will be generated by the compiler. In Figure 3 we can see a base and a derived class. The `Base` class has a proper copy constructor and a move constructor. The implementation of the copy constructor uses costly copy operations, while the move constructor uses move semantics.

In the code snippet in Figure 3, the `Derived(Derived &&)` move constructor is intended to use move semantics, and the `Base` subobject is supposed to be moved using the base class move constructor. However, because of the *if it has a name* rule explained above, the `Derived` constructor passes the (named) parameter `rhs` as an lvalue to the `Base(rhs)` constructor call, therefore the copy constructor of the `Base` subobject will be called. Even if the derived part of the object is moved, the

```
class Base
{
public:
  Base(const Base& rhs);            // copy ctor
  Base(Base&& rhs);                 // move ctor
  Base& operator=(const Base& rhs); // copy assignment
  Base& operator=(Base&& rhs);      // move assignment
  // ...
};
class Derived : public Base
{
public:
  Derived(const Derived& rhs);            // copy ctor
  Derived(Derived&& rhs);                 // move ctor
  Derived& operator=(const Derived& rhs); // copy assignment
  Derived& operator=(Derived&& rhs);      // move assignment
  // ...
};
Derived(Derived&& rhs) : Base(rhs) // wrong: rhs is an lvalue
{                                  // calls Base(const Base& rhs)
  // Derived-specific stuff
}
```

Figure 3: Derived class from Base. Wrong move constructor.

```
Derived(Derived&& rhs) : Base(std::move(rhs)) // good: rhs is rvalue
{                                              // calls Base(Base&& rhs)
  // Derived-specific stuff
}
```

Figure 4: Derived class from Base. Proper move constructor.

base subobject is copied instead of being moved. This is obviously against to the programmer's intention. As there is no syntax error in the code – just an unwanted overload resolution – no diagnostic message will be emitted by the compiler.

The correct solution is given on Figure 4. Here the `std::move` call is used to convert the named `rhs` variable to right value. We have to mention here, that `std::move` has nothing to do with the *moving* of the objects, it is merely an lvalue to rvalue conversion.

The situation in Figure 5 is similar. Here we implemented a templated swap operation, which changes the values of the two parameters using a temporary variable. In the implementation it is critical to write the `std::move` calls for the constructor and for the assignments, because the variables `a`, `b`, `tmp` all have names. Without

```
template<class T>
void swap(T& a, T& b)
{
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}
```

Figure 5: `swap` with move semantics.

```
int main()
{
  std::set<X> s1;

  s1.insert(X(1));
  s1.insert(X(2));

  std::vector<X> v1;

  v1.push_back(X(1));
  v1.push_back(X(2));

  std::vector<X> v2; v2.resize(s1.size());
  std::move(v1.begin(), v1.end(), v2.begin());
  std::move(s1.begin(), s1.end(), v2.begin());

  return 0;
}
```

Figure 6: Moving objects between containers

the lvalue to rvalue conversion using `std::move` all of them would be copied instead of being moved and the code would likely be executed slower than as assumed. As functions like `swap` are usually time critical, such performance degradations are considered as serious errors.

Given as the move semantic is relatively new in the C++ language, programmers tend to make similar mistakes. Moreover, there are situations, when mistakes related to move semantics are hidden deep in the code as we will see in the following example.

In the sample code on Figure 6 we have filled an `std::vector` container `v1` and an `std::set` container `s1` with moveable and copyable objects from the same class `X`. We then apply the `std::move` algorithm to move objects from the `v1` vector and from the `s1` set into the target container vector `v2`. The used three parameter

```
template<class InputIt, class OutputIt>
OutputIt move(InputIt first, InputIt last, OutputIt d_first)
{
  while (first != last)
  {
    *d_first++ = std::move(*first++);  // (*)
  }
  return d_first;
}
```

Figure 7: Implementation of `std::move`.

version of the `std::move` algorithm is a simple loop iteration over the interval defined by the first two parameters and applying (one parameter) `std::move` for the current element to move it to the target compiler defined by the third parameter as we seen on Figure 7. This standard algorithm is exactly the suggested tool for moving instead of copying objects from one container to an other.

Moving objects between the two vectors will work as we expected, using the implemented move assignment operator of class `X`. Surprisingly, the same algorithm will *copy* the objects from the set container to the vector using the copy assignment operator of class `X`.

To understand the reasons, first we have to consider that `std::set` is an *associative container*, i.e. objects are stored in ordered manner. For sets, the ordering key is the whole object. Therefore, modification of objects in sets are forbidden – we don't want their changed key values be inconsistent to their storing position. Thus, objects in sets behave like constants. To ensure that behavior, both `set::iterator` and `set::const_iterator` types refer to constant objects. The expression `*first++` at `(*)` on Figure 7 therefore is a reference to a constant object of type `X` which is convertible to `const X&` (copy semantics) but do not convertible to `X&&` (move semantics). As their move assignment operator is unavailable, objects in the set will be copied to the vector.

We want to emphasize, that the same algorithm, `std::move`, worked completely different way when applied to different containers. No diagnostics message was emitted, the code compiled and worked in an unintentional way. To make the situation even harder to detect, the actual copy operations happened in a standard library function.

Our method and prototype tool is analyzing such issues to detect unintentional copy operations, like those in the mentioned example.

3 Attribute-based annotations and checking

In C++11 a new feature has been introduced to able the programmers to annotate the code, and to support more sophisticated domain-specific language integration

```
template<class T>
void swap(T& a, T& b)
{
  [[move]] T tmp(std::move(a));
  [[move]] a = std::move(b);
  [[move]] b = std::move(tmp);
}
```

Figure 8: The `swap` function with annotated statements.

```
template<class T>
[[move]]
void swap(T& a, T& b)
{
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}
```

Figure 9: The annotated version of the `swap` function.

without modifying the C++ compiler [13, 14]. The new feature is called *generalized attributes* [10, 12]. Currently this is rarely used, because the lack of standard custom attributes, but it is a great extension opportunity in the language.

Most important C++ compilers, like GNU g++ and the Clang compiler (which is a C++ frontend for the LLVM [9, 11]) parse the generalized attributes, binds to the proper Abstract Syntax Tree (AST) node even if Clang displays a warning, because all generalized attributes are ignored for code generation. Even if the attributes are ignored for code generation, they are included in the abstract syntax tree, and they can be used for extension purposes. In our case, we will annotate functions and statements about the expected copy/move semantics. For example, the original code in Figure 5 can be annotated with the `[[move]]` attribute as can be seen in Figure 8.

Though, the statements are validated correctly, the annotations are redundant and every line starts with the same `[[move]]` attribute. To avoid this, the whole function can be annotated as can be seen in Figure 9. Annotating the whole function changes the default semantics of the statements inside the function – the unannotated functions have copy semantics by default.

Our validator tool use two different annotations: `[[move]]`, and `[[copy]]`. To determine the expected semantics of a statement needed the default behavior of the function (which defaults to `[[copy]]` in the most cases, but it can be overridden), and the optional statement annotation. If the statement does not have annotations, then the semantics of the statement is the same as the semantics of the function.

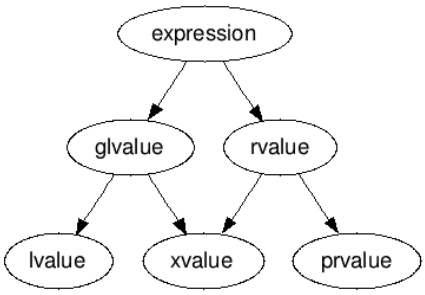

Figure 10: Value types of an expression since C++11.

If the statement contains e.g. a constructor call, then the tool checks whether the proper constructor call is made. When not the proper constructor is called, then the tool displays a diagnostic message. Furthermore, our tool validates the implementation of the move constructors. As can be seen in Figure 4, easy to make mistakes in the move constructor. As to prevent these errors, the tool sets the default semantics of the move constructors to `[[move]]` instead of `[[copy]]`.

Note that, the built-in types, like `int` and `long` do not have constructors, so the tool will allow to copy primitive types even if the `[[move]]` attribute is set. This decision is based on the fact, that there is no fundamental performance difference between moving or copying elementary built-in types.

The main algorithm of our prototype tool is based on the AST visitor mechanism provided by the `RecursiveASTVisitor` in the Clang. The visitor functions in the `RecursiveASTVisitor` class can be overridden to implement custom mechanisms. Each function in the visitor class is invoked for the specific AST nodes.

The implementation of our tool includes the overriding the specific visitor functions for the following AST nodes:

`CXXConstructorDecl:` defines a new semantic context, which can be either *copy* or *move*, depending on the constructor type and attribute (constructors are handled differently than function declarations).

`FunctionDecl:` defines a new *copy* semantic context by default, unless attribute is set (except when the function declaration is a constructor).

`CXXCtorInitializer:` used to validate constructor semantics.

`CXXOperatorCallExpr:` used to validate assignment semantics.

`CallExpr:` check the usage of the 1-parameter `std::move` function.

The role of the *semantic context* mentioned in at the `CXXConstructorDecl` and the `FunctionDecl` is to set whether validate the constructor initializers and the assignments or not. In the *move* the semantic context, our tool checks every

constructor initializers and the assignments to make sure all of them uses the move semantics. If not, then it will be reported. In the *copy* the semantic context no validation are made.

Furthermore, in the visitor functions we can use the actual AST node to gather information. The `CXXCtorInitializer` and the `CXXOperatorCallExpr` AST nodes hold information related to the move semantics. For example, starting from a `CXXOperatorCallExpr` node, we can get whether the used constructor is move constructor or not. The `CXXOperatorCallExpr` can be tighten to assignments, and we can get information about the called assignment operator.

The latest development in the tool is the `CallExpr` validation mechanism. The idea came from the fact, the 1-parameter `std::move` is a function, with a proper return value. In order to safely move the value, the returned value of the `std::move` must satisfy the following: the returned value must be *x-value*, but must not be `const` qualified. The relationship of the value types can be seen in Figure 10. The Clang sets the x-value property for all expressions which can be moveable, however, the Clang handles the `const` qualifier orthogonally.

The `FunctionDecl` class provides an attribute enumeration feature via the `attr_begin()` and `attr_end()` functions. The Clang AST does not make a difference of the "old" attribute notation and the generalized attributes. This is a known glitch in the Clang parser.

In order to handle our attributes in a standard way, we had to patch the Clang's parser. According to the documentation [18] we added two new attributes called `CopySemaExt` and `MoveSemaExt` to the `Attr.td` file. The build process generates C++ classes from the descriptions. The reason to make separate attributes for *copy* and *move* semantics is to check for the attributes easily with `hasAttr<T>()` function.

The inserted lines can be seen in Figure 11. Moreover, we placed some "boilerplate" code into the attribute processing mechanism. This is a mandatory code to tell the Clang how the attributes must be processes during the compilation. Our new attributes are simple attributes, the only requirement is to appear in the AST with an individual AST node. The affected lines which are inserted into the `ProcessDeclAttribute()` function in the `SemaDeclAttr.cpp` file can be seen in Figure 12. In the current implementation we used an additional `sem` namespace to separate our attributes from the existing ones.

The constructor type can be determined with the `isMoveConstructor()` method in the `CXXConstructorDecl` class. If the visited constructor is a move constructor, then the *move* semantic mode is pushed. In case of the visited constructor is not a move constructor, then the *copy* semantic mode is pushed.

The only relevant information of the visited `CXXCtorInitializer` is the initializer. The structure of the initializer expression provides information about the used semantics. The returned value of the `getInit()` function is a common expression, but it can be a `CXXConstructExpression`, which is a constructor call expression. The called constructor must be a move constructor for *move* semantics mode, but only if the initialized type is not a built-in type.

The tool validates the assignments via the visited `CXXOperatorCallExpr` nodes.

```
def MoveSemaExt : Attr {
  let Spellings = [CXX11<"sem", "move">];
  let Documentation = [Undocumented];
}

def CopySemaExt : Attr {
  let Spellings = [CXX11<"sem", "copy">];
  let Documentation = [Undocumented];
}
```

Figure 11: The additional lines at the end of `Attr.td`.

```
  case AttributeList::AT_MoveSemaExt:
    handleSimpleAttribute<MoveSemaExtAttr>(S, D, Attr);
    break;
  case AttributeList::AT_CopySemaExt:
    handleSimpleAttribute<CopySemaExtAttr>(S, D, Attr);
    break;
```

Figure 12: The additional lines in the `ProcessDeclAttribute()` function.

The assignments can be filtered by checking the return value of the `getOperator()` for the value `OO_Equal`. After selecting the relevant operators, the algorithm checks the right-hand side expression of the assignment. In case of *move* semantics, this expression must be an x-value and must not be const qualified.

The summarized algorithm is the following:

1. For each visited constructor and method: push the current semantic mode, and set up a new one. The new context can be either *copy* or *move*, depending on the attributes or the default mode.

2. For each constructor initializer: check whether the initializer expression meets the current semantic mode – in case of *move* semantics we require a move constructor call for non built-in types.

3. For each operator call: in case of *move* semantics we require an x-value for right-hand side expression. In the current prototype implementation we check only when `getOperator()` returns `OO_Equal`.

In the earlier paper [3] we focused on attribute-based checks, but open source projects are not using the introduced attributes. We needed an alternate approach to apply the tool on large code base.

The first remark is related to the move constructors and the move assignments. Our algorithm handles these functions differently by providing implicit move semantic context. This means, the programmer does not have to explicitly mark

```
[[sem::move]]
void f()
{
  std::vector<X> sv(4); sv.push_back(X());
}
```

Figure 13: `push_back()` guarded by the `[[move]]` attribute

these functions to use the move semantic context. Thus, the number of the potentially required attributes is minimal. In fact, only a few special functions need the `[[move]]` attribute – for example the 3-parameter `std::move` function which improper use can been seen in Figure 6.

The second remark is a modification in the algorithm. We noticed that, the use of the 1-parameter `std::move` does the most of the work. We can gather every meaningless `std::move` calls, if the result type of the call is not moveable. This change does not affect the other parts of the algorithm, because this uses the `CallExpr` AST node. Therefore, all call expressions which calls the `std::move`, and the result value is not moveable, are reported by our tool as "Suspicious move".

The misuse of the 3-parameter `std::move` are reported by the tool, so the usage of the `[[move]]` are not required here. Hence, the usage of the `[[move]]` attribute is only needed in very special cases. For example, the usage of the `std::vector` requires some attention, because the move constructor and the move assignment operator will not be used when the `noexcept` is missing from its declarations. However, an explicit `[[move]]` attribute can be useful, for instance the code snippet in Figure 13.

4 Evaluation

In the sample code in Figure 6 we used an annotated version of `std::move` to validate the semantics. To require move semantics in `std::move` we used the `[[move]]` annotation.

After running the tool on the modified code, the tool detected the misuse of the `std::move`:

```
Copy assignment found instead of move assignment, xvalue=1, const=1
```

The error message describes the problem, and the reasons as well. In the example above, the problem with the assignment is the fact the right-hand side of the assignment cannot be moved. Even it is an x-value, but it has a constant qualifier. Therefore the assignment does not satisfy the required *move* semantics.

In the sample code in Figure 3 the move constructor of the `Derived` class is wrong. Our tool can detect this kind of errors, and gives a message:

```
Copy constructor found instead of move constructor
```

```
template<class T>
[[sem::move]]
void swap(T& a, T& b)
{
  T tmp(a)); // (*1)
  a = b;     // (*2)
  b = std::move(tmp);
}
```

Figure 14: `swap` with wrong move semantics.

A intentionally wrong code can be seen in Figure 14, which is a modified version of the code seen in Figure 5. If we remove the lvalue to rvalue conversion with `std::move` in the line marked with `(*1)` in Figure 14, the tool displays the expected diagnostics message:

```
Copy constructor found instead of move constructor
```

If we also remove the lvalue to rvalue conversion with `std::move` in the next line marked with `(*2)` the tool also detect the error:

```
Copy assignment found instead of move assignment, xvalue=0, const=1
```

However, the usage of the `[[move]]` attribute is decreased since the first version of out tool. In the current version, the attribute is required only in some special cases to confirm the move semantics. One of these special cases can be seen in Figure 13. That small code snippet will copy every elements in the vector, when the `noexcept` is not present at the move constructor of the `X` class. The reason is clear: the `std::vector` will not use the move constructor if it is not `noexcept` – but the copy constructor will be used instead.

As we added extra functionality to the Clang parser, we have to consider the possible increasing of parsing time. However, the overhead during the parsing was not measureable. During the check we have two additional activities: step 1: the parsing of the generalized attributes, and step 2: evaluating of the copy/move semantics for a certain operation.

Step 1 is linearly bounded by the source size. During the evaluation of step 2 we do not descend recursively through the whole call-chain, but evaluate only on the first level copy/move operations. Therefore, we evaluate all functions only once. This is also linearly bounded by the source size. Template instantiations are evaluated for each specializations. However, this is linearly bounded by the normal template specialization generation activity of the compiler.

We measured the run time of the tool on a small part of the Clang. The original compilation time was 635,13 seconds. During the second run we inserted our checker program into the toolchain. The measured time was 637,39 seconds. The overhead is barely measurable: 2,26 second, which is less than 0,4 percent.

The tool is available as a public domain software, and downloadable from [4].

5 Future work

An outcome of our research is that, the presence of additional semantic information can be useful for the compiler. In this case, the knowledge of which operations require move semantics can help to prevent the programmer to do mistakes. And these mistakes lead to huge runtime difference since the overhead of the copy operations.

In other languages, other kind of semantic information can be found. For example, our experimental programming language, called Welltype. This language is an imperative programming language with strict syntax and strong static typesystem. The offered strict syntax is designed to lower the number of errors due to misspelling, and the semantics can prevent numerous malicious constructions, such as implicit casts. Thus, the control flow can be easily followed at source level.

Also, the Welltype handles the difference between *pure* and *impure* functions. This information is stored in the compiled binary, and it is used by the dynamic loader when a program refers to an other function – for example when the program imports a function.

Furthermore, only pure functions or expressions can be used in some constructions, such as *assertions*, *pre-* and *post-conditions*. The reason is very clear: when the compiler excludes the assertions, then the behavior of the program could be different, when the expression has side-effect. However, in Welltype the expressions used in assertions must be pure expressions – which means, only pure functions can be called. This restriction can prevent serious vulnerabilities.

We are also plan to extend our automatic checking for C++ move semantics. Since our method can be integrated into the compiler itself, thus the "3rd party tool" problem can be solved. The compiler traverses the whole AST during the compilation, and can optionally execute the sufficient move semantics checking when a certain command line parameter is set.

Furthermore, we are investigating the possibilities of the comparison-analysis at language level. The first approach is to measure the depth of nested loops. The second step is to identify definite loops – it is hard to detect loops with "read-only" loop variable (the variable must be incremented/decremented in the *step* part of the loop). Note that, these problems appear in code validation process too, because the understandable loops are important in programming.

All of the features in this paper will be implemented in our experimental programming language, called Welltype. This language supports generalized attributes as well, and the attributes can be used by the programmer – the attributes are read-only at runtime, and the dynamic program loader checks their values at link time, whether are matching with the import- and export specification.

6 Conclusion

In this paper we introduced a new method to reveal the possible misuse of the C++11 move semantics. As copy operations may determine the runtime perfor-

mance of the C++ programs, unintended use of them may have serious negative effect on the runtime performance.

We shortly described the C++ move semantics, and their positive effects on the programs. The C++ move semantic is a relatively new language level enhancement for optimizing the program execution avoiding unnecessary copy operations. Unfortunately, it is easy to make mistakes when working with move semantics. Many operations intended to use move semantics, in fact, apply expensive copy operations instead. Our method targets such mistakes, making operations planned for move semantic marked explicitly by C++11 annotations and checking their implementations for unwanted copy actions.

We implemented our method as a prototype tool to detect copy/move semantic errors in C++ programs. Our tool can deal with both regular operators or functions and constructors. The execution time of our tool is linearly depends on the size of the source code. We recognized that, the functionality of our tool can be integrated into the compiler, and the overhead of detecting copy/move semantics errors can be further decreased. Furthermore, we are intended to implement the same functionality in our experimental programming language, called Welltype.

References

[1] Abrahams, David and Gurtovoy, Alexey. *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, Boston, MA, 2004.

[2] Alexandrescu, Andrei. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, MA, 2001.

[3] Baráth, Áron and Porkoláb, Zoltán. Attribute-based checking of c++ move semantics. In *Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA) 2014, Lovran, Croatia, September 19-22, 2014.*, pages 9–14, 2014.

[4] Baráth, Áron and Porkoláb, Zoltán. Move semantics checker, 2014-2015. http://baratharon.web.elte.hu/movesem.

[5] Bloch, Joshua. *Effective Java: A Programming Language Guide. The Java Series (2nd ed.)*. Addison-Wesley, 2008.

[6] Czarnecki, Krzysztof and Eisenecker, Ulrich W. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[7] Ellis, Margaret A. and Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[8] Huynh, Bach Khoa, Ju, Lei, and Roychoudhury, Abhik. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology*

and Applications Symposium (RTAS), 2011 17th IEEE, pages 203–212. IEEE, 2011.

[9] Klimek, Manuel. The Clang AST – a tutorial. LLVM Developers' Meeting, April 2013. http://llvm.org/devmtg/2013-04/klimek-slides.pdf.

[10] Kolpackov, Boris. The Clang AST – a tutorial, April 2012. http://www.codesynthesis.com/~boris/blog/2012/04/18/cxx11-generalized-attributes/.

[11] Lattner, Chris et al. Clang: a C language family frontend for LLVM, 2014. http://clang.llvm.org/.

[12] Maurer, Jens and Wong, Michael. Towards support for attributes in c++ (revision 6). Open Standards, N2761, 2008. www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf.

[13] Porkoláb, Zoltán and Sinkovics, Ábel. Domain-specific language integration with compile-time parser generator library. *ACM SIGPLAN Notices*, 46(2):137–146, 2011.

[14] Sinkovics, Ábel and Porkoláb, Zoltán. Domain-specific language integration with c++ template metaprogramming. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 32, 2012.

[15] Standard, ISO International. Iso/iec 14882:2011(e) programming language c++, 2011.

[16] Stroustrup, Bjarne. *Design and Evolution of C++*. Addison-Wesley, 1994.

[17] Stroustrup, Bjarne. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.

[18] Team, Clang. "clang" cfe internals manual, 2014. http://clang.llvm.org/docs/InternalsManual.html.

[19] Veldhuizen, Todd. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[20] Veldhuizen, Todd. Using c++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.

[21] Wilhelm, Reinhard, Engblom, Jakob, Ermedahl, Andreas, Holsti, Niklas, Thesing, Stephan, Whalley, David, Bernat, Guillem, Ferdinand, Christian, Heckmann, Reinhold, Mitra, Tulika, et al. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.