Some Translations between Object-Oriented Languages

José de Oliveira Guimarães [1]

[1]Departamento de Computação, UFSCar
São Carlos-SP, Brazil

jose@dc.ufscar.br

Abstract. *There are several aspects to consider when analyzing an object-oriented language: support for single or multiple inheritance, dynamic or static typing, definition of subtyping, and differences between subtyping and subclassing. So different are the languages that apparently it is very or even extremely difficult to translate code in one language to any other. This article shows that this is not exactly true. We show how to translate code from and to several simplified object-oriented languages. The translation schemes help us to better understand object-oriented programming.*

1. Introduction

This article explain how to do code translation between several abstract language models. Abstract models are used because it would be impossible to cope with all details of real languages. Five language models are used, which are based on C++ [Stroustrup 1991], Java [Gosling et al. 2007], Smalltalk [Goldberg and Robson 1983], Green [Guimarães 2007] [Guimarães 2006], and Oberon [Niklaus 1988].

These languages have different type systems (static/dynamic) and vary greatly in their support for polymorphism, multiply inheritance, and other language features. By providing translations between them, we hope to draw conclusions on: a) the strengths and weaknesses of each language; b) the reasons there is more software reuse in some languages than in others; and c) the real kinship among the languages.

Although abstract models are used instead of real languages, the translations allow us to draw somes conclusions on the language themselves because the models capture the essence of the real languages.

Throughout this article, we will use the Java syntax for all language models. However, the terminology we use is *mainly* that of Smalltalk: "`x.m(0, 1)`" is a *message send* and "`m(0, 1)`" is the message sent to the object referred to by `x` at runtime. Variables declared inside a class are *instance variables*. A *method signature* is composed by the method name, its parameter types, and the return value type.

The *static or compile-time type* of a variable is the type with which the variable was declared. An expression has a compile-time type in a statically-typed language, which is the type the compiler assigns to it. Whenever a class `C` inherits from a class

B, we say B is a *direct* superclass of C. If A is a direct or indirect superclass of B, then we say that A is an *indirect* superclass of C. In this article *superclass* means *direct* superclass.

The language models of this article share some features, described next. All objects are dynamically allocated and a variable whose type is a class is in fact a pointer. At runtime, the variable will refer to an object. Except in the Smalltalk-based model, basic values such as 'A', `0`, `true`, and `3.1415` are not objects and basic types (`int`, `char`, `boolean`, `float`) are not classes. All instance variables are private. A method is either public or private. There are no static methods (C++ or Java) or class methods (Smalltalk). Classes are not considered objects. A class cannot have two methods with the same name but different number/types of parameters (method overloading).

In a statically-typed language, a class B is *subtype* of a class A if an object of B can be used where an object of A is expected without causing any runtime type errors. That is, if the compile-time types of variables `aa` and `bb` are A and B, respectively, and B is a subtype of A, then the assignment `aa = bb` is type-correct. Consider a *type* as a *class* or a basic type unless stated otherwise. Note that assignments, which encompasses parameter passing, are the only statements in which the subtype definition shows up.

The language models used in the translations are described below. Consider that their syntax are equal except in those cases in which they must be obviously different. Only the important details are described in this article, the rest being ignored.

[Oberon-M] Statically-typed language with single inheritance, subtyping equivalent to subclassing. That is, class B is subtype of class A if B inherits directly or indirectly from A. This language model will be called Oberon-M. The name "Oberon" is only to remember the model characteristics. No other Oberon [Niklaus 1988] feature than those just cited is used. The same observation is valid to the other models.

[C++-M] Statically-typed language with multiple inheritance, subtyping equivalent to subclassing. That is, class B is subtype of class A if B inherits directly or indirectly from A. Suppose class D inherits from classes B and C. Then D cannot inherit a method `m` from both B and C unless `m` is defined in a direct or indirect superclass A of both B and C. Class D then inherits the same method from two different paths. In this case, the model demands that D declare a `m` method. For short, there will never be an ambiguity in a message send regarding which method to call.

A superclass method `p` can be called using "`super.p(...)`" or, in case of multiple superclasses, using the syntax "`super(B).p(...)`" in which B is the superclass that declares `p`.

[Java-M] Statically-typed language with single inheritance and Java-like interfaces. An interface is declared as a class but it only defines method signatures. An interface can inherit from any number of interfaces. A class can *implement* any number of interfaces. If a class A implements interface `I`, then A must define all methods declared in this interface.

In this language model, a type is a class or an interface. A type B is subtype of a

type A if: a) A is interface and B inherits from A (in this case, B is an interface too) or B implements A (in this case, B is a class); b) A is a class and B inherits from A; or c) B is subtype of a subtype of A.

[Smalltalk-M] Dynamically-typed language with single inheritance. Variables are declared without types. All assignments are valid. This is the only model in which basic types (`char`, `integer`, `boolean`, `float`) are considered classes. Every basic value such as ´A´, `13`, `false`, and `3.14` is a dynamically-allocated object.

[Green-M] Statically-typed language with single inheritance and a type system in which subtyping is independent from subclassing and equivalent to set inclusion of methods.

In this model, a type is different from a class. Every class *has* a type. The type of a class is the set of signatures of its public methods. Remember a method signature is composed by the method name, its parameter types, and the return value type. The type of a class B is a *subtype* of the type of a class A if B has at least all the method signatures of A; that is, `type(A)` $\subset$ `type(B)` where `type(X)` is the type of class X. For short, we say that class B is a subtype of A. Note that every subclass is a subtype but there may be a subtype that is not a subclass — it only needs to define all the method signatures found in its supertype.

2. Translation between Object-Oriented Languages

This section shows how to translate code from some language models to others. It is important to note that almost all statements and declarations are translated to themselves in the target code. This text only comments the parts that change in the translation.

Programs in Oberon-M are already programs in C++-M, Java-M, and Green-M. Single inheritance is a special case of multiply inheritance and the subtype definition of Oberon-M is equal or more restrictive that the other models.

Throughout this section we will use some definitions, which are given next. H_C`(A)`, where A is a class, is the hierarchy of A, the set of classes calculated as follows. Consider a graph G that has a vertex for each program class and there is an edge `(B, A)` if class B inherits from class A. Then H_C`(A)` is the set of classes connected to A; that is, the set of classes found in a depth-first search starting at A.

`methodsOf(A)` is the set of public methods declared in class A. Inherited ones are not included. `allMethodsOf(A)` is the set of methods of class A, including the inherited ones. `allSubclassesOf(A)` is the set of direct and indirect subclasses of class A.

In the translations that follow, sometimes it is necessary to rename a method or instance variable. Assume that the method or variable gets a new name that is not used anywhere in the program. This observation is fundamental in all algorithms of this sec-

tion.

C++-M to Oberon-M

Translation from multiple inheritance to single inheritance in statically-typed languages. Some language features that appear in the C++-M code are translated to the same code in Oberon-M: assignments, declaration of variables, declaration of instance variables, message sends to variables, creation of objects, and treatment of basic types.

The general view of this translation scheme is as follows: a multiple inheritance class hierarchy in C++-M is converted to a single inheritance hierarchy in Oberon-M. As an example, the C++-M class hierarchy of [Fig. 1 (a)] is converted to the single inheritance hierarchy `D-C-B-A` shown in [Fig. 1 (b)].[1] Then new methods are created and message sends are modified to prevent interferences between sister classes like `B` and `C`.

We are going to explain how to do the translation of a complete hierarchy at a time. That is, the translation will be made not only for a class `A` but for all classes of the set $\mathtt{H}_C\mathtt{(A)}$. The translation scheme is described by changes in the C++-M code in order to create the Oberon-M code, just like a refactoring [Fowler et al. 1999].

First, the classes of $\mathtt{H}_C\mathtt{(A)}$ are topologically ordered resulting in $\mathtt{B}_n$, ..., $\mathtt{B}_2$, $\mathtt{B}_1$ in which $\mathtt{B}_1$ has no superclass and $\mathtt{B}_n$ has no subclass. Create a single inheritance hierarchy $\mathtt{B}_n\mathtt{-...-B}_2\mathtt{-B}_1$ in Oberon-M. The topological order preserves the subclass relationship by definition: if `B` is a direct or indirect subclass of `A` in C++-M, then `B` is direct or indirect subclass of `A` in Oberon-M too. Therefore assignments need not to be changed in the translation.

The translation would be over if $\mathtt{methodsOf(B}_i\mathtt{)} \cap \mathtt{methodsOf(B}_j\mathtt{)} = \emptyset$ for $i \neq j$, $1 \leq i \leq n$. That is, there is not a public method in common among the $\mathtt{B}_i$ classes. In this case, it would be impossible that, in C++-M, "`x.m`" at runtime call a method `m` of a class and a method of a different class in Oberon-M.

However, there is a problem if the $\mathtt{B}_i$ classes have a public method in common, as in the example of [Fig. 1 (a)]. In this hierarchy, a public method `p` is declared in `A`, `B`, and `D`, which is represented by putting a small `p` besides the class name. Since `D` inherits `p` from `A` by two different paths, model C++-M demands that `p` be overridden in `D` too. Class `C` defines a method `m` that has the following statement:

```
this.p();
```

In the C++-M model, at runtime this message send may call different `p` methods: in objects of `C`, it will call method `p` of `A` since `C` does not define a `p` method (assume this). In objects of `D`, method `p` of `D` will be called because the search for `p` begins at class `D`. For short, we will use `D::p` for method `p` of class `D`.

[1] In this figure, an arrow from a class `B` to `A` means `B` inherits from `A`.

This hierarchy is converted to the single inheritance hierarchy `D-C-B-A` in Oberon-M shown in [Fig. 1 (b)]. Then in objects of `C` the message send `this.p()` in `C::m` will call method `B::p` at runtime. This is not the semantics of the original C++-M code, which calls method `A::p`.

The problem is that class `B` in the Oberon-M code was introduced between `C` and `A`. To correct this we have to change all message sends to `this` when the method is public. Suppose there is a message send `this.p(...)` inside some method of a class `X`. Change this to `this.X_p(...)`, where `X_p` is a name that does not appear anywhere in the program. At translation time, in the C++-M code, do a search for method `p` starting at class `X`. The search continues in the superclass of `X`, superclass of superclass, and so on till the method is found in class `Y`, which may be `X`, the first class searched. In the Oberon-M code, rename the method found to `private_p` and move it to the private part. Create a public method called `p` that just calls `private_p`. Create a method `X_p` equal to `p`. This will be the "`p`" method of class `X`, the method that will not be disturbed by the introduction of an alien superclass between `X` and `Y` (as superclass `B` was introduced between `C` and `A` bringing with it a method `p` that changed the semantics of the message send `this.p()` in class `C`). See [Fig. 1 (b)] for an example where `X` is `C` and `Y` is `A`. In the Figure, an arrow between methods means that a method just calls the other. For example, the only statement of method `A::p` is a call to method `private_p` passing to it all of its real arguments.

In the C++-M code, now go down in the subclasses of `X` and do the same as above for each method `p` found: when a method `p` is found in a direct or indirect subclass `Z` of `X`, rename it to `private_p` and move it to the private part in the Oberon-M code. Create a public method called `p` that just calls `private_p`. Create a method `X_p` equal to `p`. When the algorithm goes up in the `X` hierarchy, only one class is modified. When it goes down, all subclasses are changed. Note that a) `private_p` and `X_p` are just method names that do not appear anywhere in the program and b) the search is made in the C++-M hierarchy while the modifications are made in the Oberon-M code.

Consider the example of [Fig. 1 (b)] after the modification described above. In objects of `D`, the message send `this.C_p()` in method `C::m` will call at runtime method `D::C_p`, which just calls `D::p`. In objects of `C`, the message send `this.C_p()` in method `C::m` will call method `A::C_p`, which just calls `A::p`. In both cases the semantics of the original C++-M code is preserved.

There is another problem with the example of [Fig. 1 (a)]. Suppose classes `A` and `B` define a method `t` (not shown in the Figure). In the C++-M code, a statement `super.t()` in class `C` will call method `A::t` at runtime. After the translation to Oberon-M, the hierarchy becomes `D-C-B-A` and `super.t()` in `C` will call `B::t`, changing the original C++-M meaning of the message send. This problem is solved by creating two new methods in class `A`: `private_t` and `A_t`. The original method `A::t` is renamed to `private_t` and moved to the private part of the class. Methods `t` and

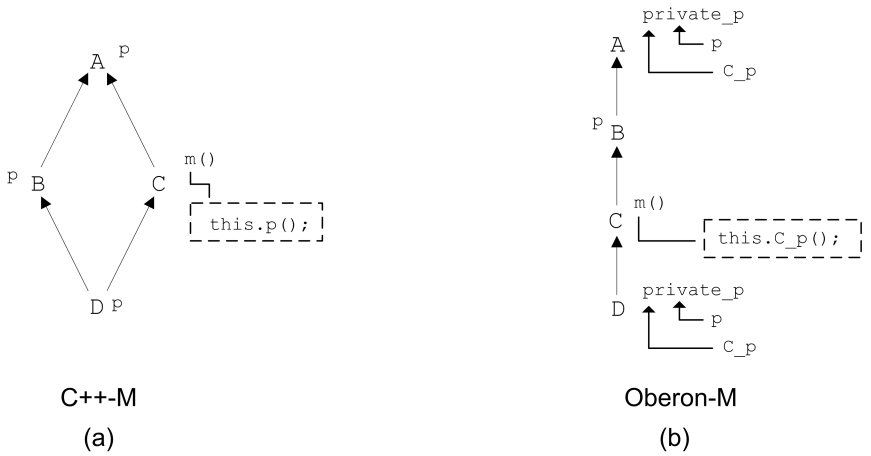

Figure 1. Translation of a C++-M hierarchy to Oberon-M

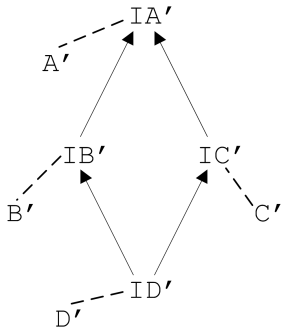

Figure 2. Interface hierarchy in Java-M that parallels that of the C++-M code

`A_t` are created in the public part. These methods just call method `private_t` passing its real parameters to it. Note that `private_t` and `A_t` are just method names that do not appear anywhere in the program. Now message send `super.t()` in `C` is changed to `super.A_t()`. Since there is no other method called `A_t` in the program, the correct method will be called.

C++-M to Java-M

Translation from multiple inheritance language to language with single inheritance with interfaces. There are two ways of doing this translation. The first and easy way is to convert C++-M to Oberon-M. The resulting code will be in Java-M too. But this destroys any multiple inheritance hierarchies found in the original code. The second way is to further convert the Oberon-M code to a model called ClassMorph-M described elsewhere [Guimarães 2008]. Model ClassMorph-M is essentially the Green-M model

but without inheritance.

After converting the C++-M code to Oberon-M and then ClassMorph-M, we create a hierarchy of Java-M interfaces to mirror those of the original C++-M code. Let us detail this.

For each program class A in C++-M, the Java-M code has an interface and a class : an interface IA′ with all the public methods of A, including those inherited, and a class A′ that implements IA′. The classes of the Java-M code, such as A′, are obtained by first converting all classes of the C++-M code to Oberon-M and then converting all classes to ClassMorph-M. Each of the resulting classes is a self-contained class without superclass (ClassMorph-M does not support inheritance). Interface IA′ is more easier to build: just collect in a set all public methods of the C++-M class A. Make IA′ inherit from interface IX′ if A inherits from X in C++-M. This creates an interface hierarchy in Java-M that is parallel to the class hierarchy of the C++-M code. Make A′ implement IA′ — this will never lead to compile-time errors because A′ has all the methods of the C++-M class A and all its direct and indirect superclasses. See an example in [Fig. 2]. A variable declared with type A in C++-M is declared with type IA′ in Java-M. An expression `new A(...)` to create an object in C++-M is translated to `new A′(...)` in Java-M.

If X is supertype of Y in C++-M, will X′ and IX′ be supertypes of Y′ and IY′ in Java-M, respectively ? Clearly X′ is not a supertype of Y′. In fact, no class has any superclass and therefore no class has any supertype that is a class. But IX′ is supertype of IY′ for the interface hierarchy mirrors the class hierarchy of C++-M.

Consider an assignment `x = y` in C++-M in which the static types of x and y are X and Y. Then X is a direct or indirect superclass of Y. In the Java-M code, the types of x and y will be IX′ and IY′ with IX′ supertype of IY′ — the subtype relationships of C++-M are preserved in Java-M. An object creation "`new A(...)`" has static type A in C++-M and is translated to "`new A′(...)`" in Java-M, which has static type A′. If A is a subtype of X in the C++-M code, IA′ is a subtype of IX′ in Java-M. Since A′ is subtype of IA′, A′ is subtype of IX′ too. This means the translation of object creation to Java-M does not introduce any type errors.

C++-M to Green-M

Translation from multiple inheritance language to a language supporting single inheritance with subtyping based in set inclusion of methods.

The translation here is very similar to that from C++-M to Java-M. The first alternative is to convert the C++-M code to Oberon-M. The resulting code is in Green-M too. The second alternative is to flatten every class A of C++-M into a class A′ without a superclass in Green-M. This is made as in the translation from C++-M to Java-M. However, it is not necessary to create Green-M classes IA′ corresponding to the interfaces

IA′ of Java-M. These interfaces were created to preserve in Java-M the type hierarchy of C++-M.

This is not necessary in Green-M: if class X is supertype of class Y in C++-M, then Y must inherit directly or indirectly from X. Therefore, Y′ has at least the public method signatures of X′ and, by the Green-M definition of subtyping, X′ is a supertype of Y′.

Smalltalk-M to Green-M

Translation from dynamically-typed language to language supporting single inheritance with subtyping based on set inclusion of methods.

A class A in Smalltalk-M is translated to class A′ in Green-M. If A inherits from class B, then A′ inherits from B′. Variables, which are typeless in Smalltalk-M, are declared with type Any in the Green-M code. The same applies to return value type of methods. Consider that Any is a class without methods and therefore supertype of every other class.[2] Therefore every variable or expression in the translated code has type Any.

For each method m of A in Smalltalk-M, create in the Green-M code an abstract class Method_m_k where k is the number of parameters of m. This class has just one abstract method:

```
Any m(Any x1, Any x2, ..., Any xk)
```

A method call `x.m(x1, x2, ..., xk)` in Smalltalk-M is translated to
`((Method_m_k ) x).m(x1, x2, ..., xk)`
in Green-M. First x is cast to Method_m_k and then the message is sent.

Class Method_m_k with its single method is necessary to call method m in Green-M. Since the compile-time type of x is Any in Green-M, a methodless class, no message can be sent directly to x. Note that the real arguments of the message send, x_i, have type Any and the formal parameters of the method m of class Method_m_k also have type Any.

Does this scheme also work with basic values such as 1, 'A', or 3.14 ? Not yet, since a message send a + 1 in Smalltalk-M would be translated to a + 1 in Green-M and this would be a sum of a variable of type Any with 1. To solve this problem, create in Green-M a wrapper class for each of the basic types int, char, boolean, and float. The wrapper classes have names Int, Char, Boolean, and Float. Every wrapper class stores a value of the corresponding type and has methods get and set to retrieve and set the value. Every wrapper class has methods corresponding to the operations the basic type supports. For example, class Int has a method to add two Int

[2] In the real language Green, there is a class Any that is inherited by any class that does not inherit from any other

objects, `Any plus (Any other )`.

For each wrapper class, there should be created classes of the kind `Method_m_k` as if the wrapper classes were in the Smalltalk-M code. Then there is a class `Method_plus_1` with method `Any plus(Any other)`.

We are going to make Green-M simulates the basic values and types of Smalltalk-M. First, each basic value literal such as `1`, `'A'`, or `3.14` of the Smalltalk-M code is translated to an object creation in Green-M using the appropriate wrapper class. For example, "`1`" in Smalltalk-M is translated to "`new Int(1)`" in Green-M.

Second, a message send "`a op b`" in Smalltalk-M, where `op` is an arithmetical or logical operator (`+`, `-`, `*`, ... `and`, `or`, `not`, ...), is translated to

```
((Method_opname_1 ) a).opname(b)
```

in Green-M, where `opname` is the name in English of the operator, the same name used in the methods of the wrapper classes. For example, `a + b` in Smalltalk-M is translated to

```
((Method_plus_1 ) a).plus(b)
```

in Green-M.

The same mechanism is used with unary operators. Using this translation scheme, the produced Green-M code is obviously type correct (for everything has type `Any`). If a method `X::m` is called in Smalltalk-M at runtime because of message send "`x.m()`", then the same method will be called in the translated Green-M code. After all, the runtime search for a method is the same in both languages and the translation scheme does not change the class hierarchies.

This scheme only introduces types and type casts in the code to make it compatible to the Green-M type system. The semantics of the code is not changed in any way.

Green-M to Java-M

Translation from language supporting single inheritance with subtyping based in set inclusion of methods to language with single inheritance with interfaces.

For each method of each class in Green-M, create an interface in Java-M. From a Green-M method

$R\ m(T_1\ x_1,\ T_2\ x_2,\ \ldots,\ T_k\ x_k)$

interface $Method\_m\_T_1\_T_2\_\ldots\_T_k\_\_\_R$ is created in Java-M. This interface declares a single method whose signature is equal to `m`.

A class `A` in Green-M is translated to class `A'` in Java-M. If `A` inherits from class `B`, then `A'` inherits from `B'`. Class `A'` implements interfaces `Method_m_...` corresponding to all of its methods, including the inherited ones.

Both Green-M and Java-M do not consider basic types as classes: the semantics of basic types and values is the same in the two languages. Then expressions, literals, and types related to basic types in Green-M are translated to themselves in Java-M.

However, every variable in Green-M whose type is a non-basic type (a class) will have type `Any` in the translated Java-M code. The same applies to return value of methods. Class `Any` is created by the translator and has no methods.

A message send

`x.m(`e_1`, `e_2`, ..., `e_k`)`

in Green-M is translated to

`((Method_m_`T_1`_`T_2`_..._`T_k`__R) x).m(`e_1`, `e_2`, ..., `e_k`)`

in Java-M, assuming that method `m` was declared as

`R m(`T_1 x_1`, `T_2 x_2`, ..., `T_k x_k`)`

Does this scheme works ? Does it introduces any errors ? Let us see that. This scheme produces a type correct Java-M code since all types but the basic ones are converted to `Any`. The runtime search for a method that occurs after a message is sent is equal in Java-M and Green-M. Therefore, the semantics of the source and target codes are equal — message sends are not changed in the translation, only type casts are introduced in Java-M.

There is another way to translate Green-M to Java-M. First, one can collect all subtype information in all Green-M code. Whenever, in the Green-M code, an object of type `Y` is used where an object of type `X` is expected, we register that `Y` is a subtype of `X`. That is, there is an assignment (which includes parameter passing) `x = y` in which the types of `x` and `y` are `X` and `Y`, respectively. Of course, `y` could be an expression.

Now the translation is as follows: for each class `A` in the Green-M code, create a class `A′` and an interface `IA′` in the Java-M code. Class `A′` has all methods defined in class `A`. If class `A` inherits from class `B`, then class `A′` inherits from class `B′`. And class `A′` implements interface `IA′`, being therefore a subtype of it.

Interface `IA′` declares all the public methods of `A′`, including the inherited ones, and inherits from all the interfaces corresponding to supertypes of `A` in the Green-M code (supertypes, not only the superclasses). That is, if { `C`, `D`, `E` } is the set of supertypes of `A`, then interface `IA′` inherits from interfaces `IC′`, `ID′`, and `IE′`. These subtype relationships are those collected in the Green-M code.

A variable declared with type `A` in Green-M will have type `IA′` in Java-M. The creation of objects in Java-M uses the prime classes. For example, "`new A()`" in the Green-M code is translated to "`new A′()`" in Java-M.

Subtype relationship in the Green-M code is not preserved in the translated Java-M code. That is, there may be a class `X` supertype of `Y` in Green-M and `IX′` is not supertype of `IY′` in Java-M. But this will only happen if `X` is not effectively used as a supertype of `Y`

in the Green-M code. If it is, this subtype relationship is registered by the translator and used to set the inheritance of the interfaces in Java-M. For short, every effective subtype in Green-M is a subtype in Java-M.

If a message send `x.m(...)` calls at runtime method `X::m` in Green-M, which method does it call in Java-M ? Every class in Green-M is translated to a similar class in Java-M and inheritance relationships are preserved in the translation. Since the runtime search for a method is the same in both languages, the method called at runtime will be the same, `X::m`.

3. Conclusion

Some of the models described in this article offer more support to software reuse than others. For example, a method "`m`" that takes a formal parameter of class `A` in Oberon-M/C++-M can receive as argument at runtime an object of class `A` or any of its subclasses. In Java-M, the argument to `m` can be an object of class `A` or its subclasses or, if this class were an interface, implements directly or indirectly interface `A`. However, a class has to declare explicity that it implements an interface, just like a declaration of inheritance. In Green-M, the argument can be an object of any class that declares the same methods as `A`. This includes `A` and all of its subclasses plus possibly many more classes that do not inherit from `A` (but that are *subtypes* of this class). The code of `m` can be reused in more occasions than in Oberon-M/C++-M/Java-M. It is not necessary to declare explicity that a class is *subtype* of `A`. In Smalltalk-M, method `m` is declared without the formal parameter type. Any object can be passed as a parameter. At runtime there will be no error if the object has the methods corresponding to the messages sent to it. The object can have fewer methods than those declared in class `A` because of the runtime behavior of the statements of method `m`. If `A` declares a public method `p` and the real argument is an object which does not have a `p` method, no error will occur if message `p` is not sent to the argument at runtime. Therefore objects of more classes than be passed as arguments to the method, which can be reused in more occasions than in Green-M. We can conclude that software reuse is higher in some models than in the others in the following order: Smalltalk-M, Green-M, Java-M/Oberon-M/C++-M.

The C++-M model offers multiple inheritance, which allows a class inherit methods and instance variables from several superclasses. It is not obvious how a convoluted class hierarchy with numerous multiple inheritances can be translated into a single inheritance hierarchy. The problem is that sister classes in the multiple inheritance hierarchy can become subclass and superclass in the resulted single inheritance hierarchy. This brings semantic and syntactic problems, although problems that can be solved.

Finally, all models but Smalltalk-M are statically-typed. It is not obvious how to translate a dynamically-typed language to a statically-typed one. For all of the differences cited above, it is a little bit surprising that code in some models can be translated into others while still retaining some original code characteristics. In fact, every model can

be translated into any other. This is shown in an unabridged unpublished version of this article [Guimarães 2008].

The objective of this article is clearly not to offer some new tools for compiler designers. It would be too cumbersome to use our translation schemes for code generation in real compilers. However, the first ideas about language to language translations came about when building the Green Compiler [Guimarães 2007] [Guimarães 2003]. We needed a fast way of translating Green and we chose to translate it to Java.

The main objective of this article is to improve the understanding of the several models of object-oriented languages. By knowing the details of them and how to translate one language to other, we can grasp all the subtlenesses of every model. We conclude with comments on the languages and their translations.

Java-M and Green-M do not offer multiple inheritance. But since they support multiple subtyping, they cannot be easily translated to Oberon-M. The translation is so difficult as that from C++-M to Oberon-M. The runtime search for a method after a message send is the same in Java-M, Green-M, and Oberon-M because all of these languages support only single inheritance. However, this does not make it easy to preserve subtyping in the translation because types are a compile-time feature.

Oberon-M is clearly a subset of Java-M, Green-M, and C++-M. Java-M can be made a subset of Green-M by a) changing every interface into a Green-M abstract class and b) removing any implementations of an interface by a class. That is, given

```
class A implements I, J { ... }
```

in Java-M, just remove `implements I, J`;

It is easy to translate every single inheritance model to Smalltalk-M because this language model has the most liberal type system. And because the runtime search for a method after a message send is the same in all single inheritance models. The translations are not shown in this paper but can be found in [Guimarães 2008].

The translation schemes can be used as techniques to implement in a real language some features it does not support. In fact, some or part of the schemes can originate idioms, which are Design Patterns [Gamma et al. 1994] specific to a language. The available compiler of Green, the real language, produces Java as output. Everything is mapped to Java constructions, including the exception handling system, which is completely object-oriented. This translation originates an idiom [Guimarães 2001] for exception treatment that simulates in Java/C++ the object-oriented features of the exception system of Green.

Although both C++-M and Java-M support multiple subtyping, there is no easy way to translate C++-M to Java-M. Multiple inheritance of C++-M means multiple inheritance of code and multiple subtyping. Only the last feature is supported by Java-M. This leads naturally to the translation scheme exemplified by [Fig. 2], in which an interface hierarchy in Java-M is created for every C++-M class hierarchy. However, this scheme

also demands the flattening of every C++-M class into a Java-M class without superclass.

The translation from C++-M to Green-M shows the power of polymorphism in Green-M: every C++-M class is flattened into a Green-M class without superclass and that is all. No class/interface needs to be created because of the Green-M subtype definition.

The translation from Smalltalk-M to Green-M only demands changes in variable/return value types and the introduction of some type casts. This shows polymorphism in Green-M is not much less powerful than that of Smalltalk. However, in the translated Green-M code all types are `Any`. It could not be different since the runtime types of a variable in Smalltalk-M are not computable. This scheme requires the creation of a class for each method/number of parameters of the program. However, the translator does not need to examine the entire program before starting the translation, as in the second alternative of converting Green-M to Java-M. Classes like `Method_m_k` can be created as the message sends are found by the translator. All the runtime type checking of Smalltalk-M is translated to just a cast to a class, as in

$$\texttt{((Method\_m\_k) x).m(}x_1\texttt{, }x_2\texttt{, ..., }x_k\texttt{)}$$

The translation from Green-M to Java-M shows how different are the subtype definitions in these languages. Although both support only single inheritance, the differences are profound in their support for polymorphism. In Java-M, a subtype must be explicitly declared. But that is not always easy to do. For example, if class `B` inherits from class `A` and we want it to be subtype of class `C`, we cannot. To achieve this or something equivalent, it is necessary to change several classes and the code that uses these classes, which may be spread in the whole program. It would probably be necessary to create some interfaces. It can be worse. If we do not have access to the source code of classes `A` and `B`, it is impossible to make `B` subtype of `C`.

Green-M does not have these limitations. A class is automatically made subtype of every other class that declares a subset of its public methods. The contrast Green-M-automatic and Java-M-declared subtyping shows itself in the two translations schemes presented. In the first, the translator declares all variables in Java-M with type `Any`, except those variables that have a basic type in Green-M. This radical approach transfer to runtime all type-checking — but this is not unsafe since Green-M is statically-typed and the translation guarantees that a method will be found at runtime.

In the second scheme, the translator has to collect subtyping information in the whole program before starting the translation. Hierarchies of interfaces in Java-M mimics the real hierarchies of subtypes in the Green-M code. A variable in Java-M has a type which is closely linked to its type in Green-M. Then this scheme offers a better representation in Java-M of the Green-M code. The price of this is the need of analyzing the whole program before the translation to collect subtyping information. Subtyping, which is implicit in Green-M, is made explicit in the generated Java-M code by this analyzing step.

References

America, P; Linden, F. V. D. (1991). “A parallel Object-Oriented Language with Inheritance and Subtyping”; Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, 161-168.

Fowler, M.; Beck k.; Brant, Opdyke, W.; Roberts, D. (1999). “Refactoring: Improving the Design of Existing Code”; Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). “Design Patterns: Elements of Reusable Object-Oriented Software”; Addison-Wesley, Reading, MA.

Goldberg, A., Robson, D. (1983). “Smalltalk-80: The Language and its Implementation”; Addison-Wesley, Reading, MA.

Gosling, J., Joy, B., Steele, G., Bracha, G. (2005). “The Java Language Specification”; third edition, Prentice Hall PTR. Available at http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

Guimarães, J. (2001). “An Idiom for Exception Treatment in C++ and Java”; Proc. V Simpósio Brasileiro de Linguagens de Programação (Brazilian Symposium of Programming Languages).

Guimarães, J. (2008). “On Translation between Object-Oriented Languages”; Available at http://www.dc.ufscar.br/~jose/green/green.htm.

Guimarães, J. (2007) “The Green language”; Available at http://www.dc.ufscar.br/~jose/green/green.htm.

Guimarães, J. (2006). “The Green language”; Computer Languages, Systems, & Structures, 32, 4, 203-215.

Guimarães, J. (2003). “Experiences in Building a Compiler for an Object-Oriented Language”; SIGPLAN Notices, 38, 4, 25-33.

Wirth, N. (1988). “The Programming Language Oberon”; Software, Practice & Experience 18, 7, 671-690.

Stroustrup, B. (1991). “The C++ Programming Language”; third edition, Addison Wesley, Reading, MA.