On Translation between Object-Oriented Languages

José de Oliveira Guimarães
(UFSCar, Brazil,
jose@dc.ufscar.br)

Abstract There are several aspects to consider when analyzing an object-based or object-oriented language: support for single or multiple inheritance, dynamic or static typing, definition of subtyping, differences between subtyping and subclassing, and support for inheritance. 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-based and object-oriented languages. The translation schemes help us to better understand object-oriented programming.

Key Words: object-oriented languages, Green, Smalltalk, Java, C++, type system, polymorphism
Category: D.3.2, J.5

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. Six 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], ClassMorph, and Oberon [Niklaus 1988]. ClassMorph is a fictitious object-based language (without inheritance) but supporting polymorphism.

Throughout this article, we will use the Java syntax for all language models. A class `Store` in this language is shown in [Fig. 1]. Keyword `this` is the equivalent of `self` in other object-oriented languages such as Smalltalk. A class `B` that inherits class `A` is declared as

```
class B extends A { ... }
```

However, the terminology we use is *mainly* that of Smalltalk: "`x.m(0, 1)`" is a *message send*. "`m(0, 1)`" is the message sent to the object referred to by `x` at runtime. Variables declared inside a class, such as `n` of `Store`, are *instance variables*. A *method signature* is composed by the method name, its parameter types, and the return value type. For example, the method signatures of `set` and `get` of class `Store` [Fig. 1] are:

```
void set(int)
int get()
```

```
class Store {
   public void set( int n ) {
      this.n = n;
   }
   public int get() {
      return this.n;
   }
   private int n;
}
```

Figure 1: Example of a class in Java

The *declared or compile-time type* of a variable is the type with which the variable was declared. For example, the declared type of instance variable `n` is `int`. An expression have a compile-time type in a statically-typed language, which is the type the compiler assigns to it. For example, the compile-time type of `0 + 1` is `int` in Java. 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. A *hierarchy* is a set S of classes such that for any two classes `A`, `B` $\in$ S, one of two things happens: a) one of the classes is a direct or indirect superclass of the other or b) `A` and `B` share a common direct or indirect subclass or direct or indirect superclass. We say S is the hierarchy of a class `A` if `A` $\in$ S. Given two hierarchies S and R, either S $\cup$ R $= \emptyset$ or S $=$ R.

The language models of this article share some features, described next.

- There are four basic types: `int`, `char`, `boolean`, `float`.
- 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 are not classes.
- In a message send `x.m(a, b)` the compiler checks whether the declared type of `x` is a class and whether this class or any of its superclasses (direct or indirect) defines a method `m` that can accept `a` and `b` as parameters. Assume the declared type of `x` is `X`, the language is statically-typed, and supports only single inheritance. There is a *compile-time search*: first the method is searched for in class `X`, then in the superclass of `X` (if any), the superclass of superclass, and so on. This algorithm is easily generalized to multiple inheritance (as will be seen, there will not be any ambiguity in which method `m` should be chosen, if the hierarchy has more than one).

– In a message send `x.m(a, b)`, there is a *runtime search* for method `m`. Assume the language supports only single inheritance. Note that it does not matter whether the language is static or dynamically-typed. At runtime, suppose `x` refer to an object of a class `A`, which must be a *subtype* of the declared type of `x` (a *subtype* is defined in the following paragraphs). The runtime system searches for a method `m` in the public methods of class `A`. If none is found, the search continues in the superclass of `A`, in the superclass of the superclass, and so on. This algorithm is easily generalized to multiple inheritance (there will be no ambiguity in which method to choose, if there is more than one).

– When the compiler finds a message send `this.m(a, b)` inside a method of a class `A`, it checks whether there is in `A` a private method `m` that can receive `a` and `b` as parameters. If there is one, this method will be called at runtime — there is no runtime search for a method.

  If no private method `m` is found in `A`, the search continues in the public methods of `A`, then in the public methods of the superclass of `A`, and so on. If no method is found, the compiler signals an error. At runtime, suppose `this` refer to an object of a class `B`, which must necessarily be a *subclass* of `A` — a subclass, never a subtype as defined in the following paragraphs. To execute statement `this.m(a, b)`, the runtime system searches for a method `m` that can receive `a` and `b` as parameters in the public methods of `B`. The search continues in the superclass of `B`, the superclass of superclass, and so on. These algorithms are also easily generalized to multiple inheritance.

  Note that this compile and runtime searches are made in statically and dynamically-typed language models.

– When the compiler finds a message send `super.m(a, b)` inside a method of a class `A`, it searches in the superclass of `A` for a public method `m` that can receive `a` and `b` as parameters. If none is found, the search continues in the superclass of the superclass, and so on. At runtime no search is necessary: the method to be called is defined at compile-time. This search is made in statically and dynamically-typed language models.

– An abstract class, declared as `abstract class A { ... }` declares method signatures but it does not define any method bodies. This is different from most languages in which an abstract class may have some or all methods with bodies.

– All instance variables are private and therefore only methods of the class can access them. A method is either public or private.

– There are no static methods (C++ or Java) or class methods (Smalltalk). A method is only called when a message is sent to an object. Classes are not considered objects.

– A class cannot have two methods with the same name but different number/types of parameters. This is not a limitation, since before the translation the overloaded methods can be renamed.

– A cast from an expression `expr` to type `A` is make with the following syntax:

```
x = (A ) expr
```

The resulting value is assigned to `x`. There may be a runtime error because `expr` may not be convertible to `A`. For example, `expr` may refer at runtime to an object of class `B`. If `B` is not a subtype of `A`, there is an error.

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.

1. 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 are used. The same observation is valid to the other models.

2. 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 `G` inherits from classes `E` and `F` such that these classes have no common superclass — see an example in [Fig. 2 (a)]. Then no public method should be inherited by both `E` and `F`. There should be no name conflict between inherited methods. This is not a limitation, since conflicting methods can always be renamed before the translation.

   Suppose class `A` of [Fig. 2 (b)] defines a method `m`. Then `D` inherits `m` from `A` by two different paths: `ABD` and `ACD`. In this case, we demand class `D` defines a method `m` to remove the ambiguity. In objects of `D`, there will be only one set of instance variables of `A`.

Figure 2: Two examples of multiple inheritance

In classes with a single superclass, as `B` or `C` of [Fig. 2 (b)], a method `m` of the superclass can be called using the syntax "`super.m(...)`". If there is more than one superclass, its name should be cited in the message send to `super`: in class `D`, `super(B).m(...)` will call method `m` of `B` (which may be inherited from `A`).

This language model will be called C++-M.

3. 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 using the syntax:

   ```
   class A implements I, J, K { ... /* class body */ }
   ```

   If a class `A` implements interface `I`, as in this example, then `A` must define all methods declared in this interface. Otherwise there is a compile-time error. An interface may be the return value type of a method or the declared type of a variable. But since interfaces are not classes, they cannot be used to create objects with "`new`".

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

   - `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);
   - `A` is a class and `B` inherits from `A`;
   - `B` is subtype of a subtype of `A`.

   This language model will be called Java-M.

4. Dynamically-typed language with single inheritance. Variables are declared but without types. All assignments are valid. This is the only model in which basic types (`char`, `integer`, `boolean`, ...) are considered classes. Then a basic value such as `'A'`, `0`, `true`, and `3.1415` is considered an object and it must be dynamically allocated.

   Every method should return a value. If a method does not return anything, it automatically returns `this`.

   This language model will be called Smalltalk-M.

5. 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. For example, the type of class `Store` of [Fig. 1], `type(Store)`, is {`void put(int)`, `int get()`}. 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.

   This language model will be called Green-M.

6. Statically-typed object-based language with subtyping equivalent to set inclusion of methods as in Green-M. This is not an object-oriented language as it does not support inheritance. But it does support classes and uses a subtype definition equal to that of Green-M.

   This language model will be called ClassMorph-M. Language ClassMorph is a fictitious language.

In the translation between models, we are going to consider relevant only some specific language features:

1. assignments. This encompasses the passing of real arguments in method calls. When a real method argument is set to a formal method argument, there is an implicit assignment;

2. inheritance;

3. declaration of variables and method parameters;

4. declaration of instance variables;
5. declaration of public and private methods;
6. message sends to `this` as in "`this.m()`" in which `m` is a method name. There are two cases to consider: the corresponding method `m` is public or private;
7. message sends to variables as in "`x.m()`" in which `x` is a variable. This case encompasses also message sends to variables accessed through `this`: "`this.x.m()`";
8. message sends to super as in "`super.m()`";
9. creation of an object as in "`a = new A()`", in which an object of class `A` is created;
10. treatment of basic types (`int`, `char`, `boolean`, `float`). In Smalltalk-M, basic types are classes and therefore basic values (`1`, `'A'`, `true`, etc) are objects. In all other language models, basic values are not objects.

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 changes in the translation.

The set of programs in some languages are a subset of the set of programs in other languages. Therefore, no translation is really necessary. The conversions that match this criteria are:

1. from Oberon-M to C++-M, because single inheritance is a special case of multiple inheritance and the subtype definitions are equal in both languages;
2. from Oberon-M to Java-M or Green-M. These languages support only single inheritance. Since the subtype definition of Java-M and Green-M are more encompassing than that of Oberon-M, an assignment in Oberon-M will remain type-correct in Java-M or Green-M. Then every statement in Oberon-M is also a statement in Java-M or Green-M;
3. from ClassMorph-M to Green-M. No inheritance (ClassMorph) is a special case of single inheritance (Green-M). Since the subtype definition of both models are equal, every statement of ClassMorph-M is also a statement of Green-M.

Throughout this section we will use some definitions in the algorithms, which are given below.

– $\texttt{H}_C\texttt{(A)}$, where `A` is a class, is the hierarchy of `A`, the set of classes calculated as follows. Consider each class of the program a vertex in an undirected graph. There is an edge between two classes if one inherits from the other. Then $\texttt{H}_c\texttt{(A)}$ is the set of all 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 methods declared in class `A`. Inherited ones are not included.

– `allMethodsOf(A)` is the set of methods of class `A`, including the inherited ones.

– `superclassOf(A)` is the set of direct superclasses of class `A`. In an single inheritance language, `superclassOf(A)` returns a single class or `nil`.

– `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 section.

Oberon-M to ClassMorph-M

This is a translation from a single inheritance statically-typed model to a language without inheritance but with polymorphism. We are going to translate one class at a time. Let `A` be a class in the Oberon-M model we are going to translate to a class $\texttt{A}'$ in ClassMorph-M — the class name could be the same in both models. Different names are used only to make the text clear. The behavior of class `A` in Oberon-M should be the same as the behavior of $\texttt{A}'$ in ClassMorph-M.

It is not necessary to consider in the translation all the aspects cited in the end of [Section 1]. In particular, assignments, declaration of variables, creation of objects, and treatment of basic types are the same in Oberon-M and ClassMorph-M. This will soon be justified.

A general view of the translation is as follows: class `A` and its direct and indirect superclasses are collapsed into class $\texttt{A}'$. Then class $\texttt{A}'$ has all instance variables and methods of `A` (public and private) and all of its superclasses. All instance variables and private methods are renamed as are all methods that are overridden in subclasses. The translation is explained in details now.

Class $\texttt{A}'$ has all the private methods and instance variables of class `A` and all its direct and indirect superclasses. These methods and variables are renamed in order to avoid name clashes. The set of public methods of $\texttt{A}'$ is calculated by the following algorithm. The result is put in set `S`.

```
Let S be the set of public methods of A′, initially empty.
S = set of public methods of A
X = superclassOf(A)
while X <> nil do
    begin
    add to S all public methods of X not yet in S
    rename those public methods of X already in S and
    add them to the private part of A′
    X = superclassOf(X)
end
```

As stated previously, if a class does not have a superclass, then "`superclassOf(X)`" returns `nil`.

In a message send `x.m()` in which `x` refers to an `A` object of Oberon-M, the method `m` called will be searched at runtime starting in the class of the object, `A`. If `m` is not found in `A`, the search continues in the superclass, the superclass of the superclass, and so on. In ClassMorph-M, the method called at runtime by `x.m()` should be the same as in Oberon-M. This do happens because the algorithm above, executed at compile-time, is equivalent to the runtime search for a method in Oberon-M. The method called in both cases is the same. Therefore, an Oberon-M message send `x.m()` in which `m` corresponds to a public method does not need to be changed in the translation.

Exactly the same reasoning is applied in message sends to `this` when `this` refers to an `A` object in Oberon-M and the corresponding method is public. That is, we have "`this.m()`" and `m` corresponds to a public method. The runtime search in Oberon-M is converted to the compile-time search made by the algorithm above.

There are two other occasions in which a method is called in Oberon-M:

- in message sends to `this` when the corresponding method is private;
- in message sends to `super`.

In both cases, the method to be called is determined at compile-time. Then in ClassMorph-M we already know which method to call at compile-time. It is only necessary to change the name of the messages to reflect the renamed methods, when this is the case. That is, if a message send `super.m()` in class `A` calls method `m` of superclass `W` of `A`, this message send should be changed to `super.m_W()` in class `A′` of ClassMorph-M if method `m` of `W` was renamed to `super.m_W()`. As seen, the overridden public methods are renamed.

With this translation scheme, every message send to an `A` object will call the same method as the corresponding message send in class `A′`. Note the grouping of methods of `A` and its superclasses in `A′` demanded that some runtime searches

in Oberon-M are made at compile-time in ClassMorph-M. This runtime optimization is not for free: the code of methods of superclasses of `A` are tailored to `A′`. If another class inherit the same superclasses as `A`, the methods should be changed specifically to this class in ClassMorph-M. Code duplication is the price of transforming runtime searches of Oberon-M in compile-time searches in ClassMorph-M.

Why does this work ? First, variable declaration are not changed — every variable in ClassMorph-M has the same type as in Oberon-M. Second, if `X` is supertype of `Y` in Oberon-M, then `X` is a direct or indirect superclass of `Y`. This mean that `X′` is supertype of `Y′` in ClassMorph-M, in which `X′` and `Y′` are the classes in ClassMorph-M corresponding to `X` and `Y`. This occurs because the set of public methods of a class, considering the inherited ones, does not change with the translation — they are preserved by the algorithm. And because the subtype definition of ClassMorph-M is more general than that of Oberon-M. The consequences of this is that any assignment in Oberon-M is also correct in the translated ClassMorph-M code.

To do this translation by hand is an excellent way of learning the basics of object-oriented programming. In particular, it is an excellent way of learning how message sends to `this` and `super` works. This translation can be made from Green-M to Green-M — one does not need to use ClassMorph as the target language.

Java-M and Green-M to ClassMorph-M

This translation is the same as that from Oberon-M to ClassMorph-M. It works because the subtype definition of ClassMorph-M is more encompassing than that of Java-M and equal to the subtype definition of Green-M. If class `X` is supertype of class `Y` in Java-M or Green-M, then necessarily class `Y′` in ClassMorph-M has at least all public methods of class `X′`. Then `X′` is a supertype of `Y′` in ClassMorph. Consider that a class `X` in Java-M or Green-M is translated to `X′` in ClassMorph-M.

Oberon-M, Java-M, Green-M, or ClassMorph to Smalltalk-M

Translation from statically-typed languages to a dynamically-typed one.

A class `A` in the source language is translated to class `A′` in Smalltalk-M. If `B` inherits from `A`, then `B′` inherits from `A′`.

Every statement in the source language is translated to itself in Smalltalk-M. Every variable or method declaration is translated to a declaration without types in Smalltalk-M. This language has the most liberal type system, which is

dynamically-typed. We can consider that, at compile time, any type is subtype of any other. Therefore, every subtype in the source language remains subtype in Smalltalk-M.

At runtime, in every message send there is a search for a method starting in the class of the object that received the message. This search continues in the superclass, superclass of the superclass, and so on. This search is not modified by the translation since the source languages support at most single inheritance, as Smalltalk-M.

The source languages are statically-typed and the method is always found at runtime. The same happens in the code translated to Smalltalk-M, for the semantics of it is not changed.

The source languages do not consider basic types as classes. The target language does. This does not cause any problems, since every use of a basic type or value in the source languages remains exactly the same in Smalltalk-M. To understand how this works, one can study the automatic conversions of basic values in Green [Guimarães 2006] and the boxing/unboxing feature of Java [Gosling et al. 2007]. In these languages, it is as if basic values were objects. Whenever necessary, they are converted to objects using wrapper classes. For short, an expression `1` or `1 + 2` in the source language should be translated to itself in Smalltalk-M. And an assignment `x = y` in which `x` and `y` have basic types, is translated to itself in the Smalltalk-M code and the meaning of it remains the same in both languages.

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. 3 (a)] is converted to the single inheritance hierarchy `D-C-B-A`, in which `D` inherits from `C` that inherits from `B` that inherits from `A`.

This brings some problems. In this example, class `B` was introduced, in the Oberon-M code, between class `C` and its superclass `A` (in C++-M). Calls to `super` and `this` in `C` may call `B` methods in the flattened hierarchy in Oberon-M, which never happens in the original C++-M code. That is corrected by creating new methods in the Oberon-M classes that prevents any interference between two sister classes like `B` and `C`. Each of these methods play the same role as a specific

method of the class and has a name different from any other method in the program. Since the method name is different from any other, no interference among classes may arise.

The translation will be explained in details now.

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 $\mathtt{H}_C\mathtt{(A)}$.

Although the inheritance hierarchy in the C++-M code is flattened in the translation to Oberon-M, if class `B` is direct or indirect subclass of `A` in C++-M, `B` should remain direct or indirect subclass of `A` in Oberon-M. Let us study two cases, shown in [Fig. 2]. Remember that there is no name conflict between inherited methods. In case (a), in the Oberon-M code there will be a class `G` that inherits from `E` that inherits from `F`. This will be represented as `G-E-F`, a single inheritance hierarchy. Since $\mathtt{methodsOf(E)} \cap \mathtt{methodsOf(F)} = \emptyset$, no `F` method will be overridden in class `E`. Of course, it is easy to generalize this case when a class `A` inherits from $\mathtt{A}_i$, $1 \leq i \leq n$ and

1. $\mathtt{A}_i$ and $\mathtt{A}_j$ has no common direct or indirect superclass when $i \neq j$;
2. $\mathtt{allMethodsOf(A}_i\mathtt{)} \cap \mathtt{allMethodsOf(A}_j\mathtt{)} = \emptyset$ for $i \neq j$.

In [Fig. 2 (b)], we cannot simply flatten this inheritance hierarchy to `D-C-B-A` because this could change the meaning of some message sends. Let us see the example shown in [Fig. 3 (a)]: suppose class `A` defines a method `p` which is overridden in class `B`. Since `D` inherits `p` from `A` by two different paths, we demand `p` is overridden in `D` too. Class `C` defines a method `m` that has the following statement:

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

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 `D`. For short, we will use `D::p` for method `p` of class `D`.

If this hierarchy is converted to the single inheritance hierarchy `D-C-B-A` in Oberon-M, then in objects of `C` the message send `this.p()` 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 no appear anywhere in the program. Do a search for method `p` starting in 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. Rename the method found to `private_p` and move it to the private

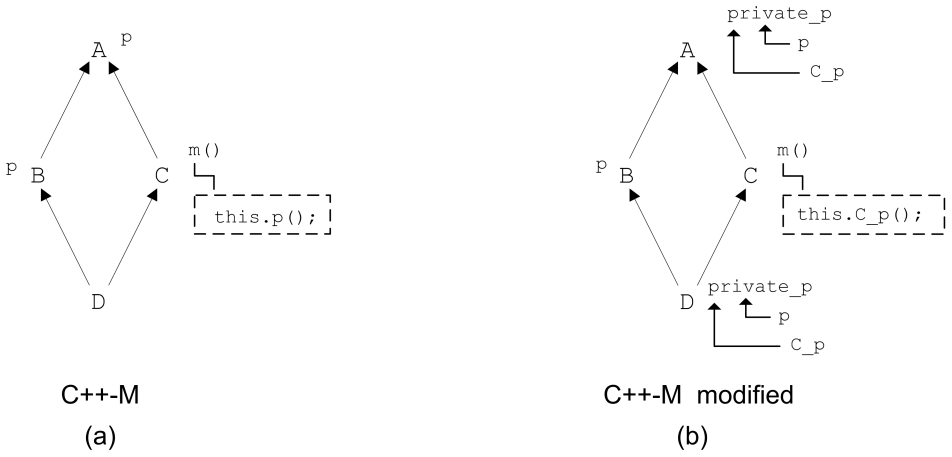

Figure 3: Modifications in the C++-M hierarchy in preparation to translate it to a single inheritance language

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`).

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. Create a public method called `p` that just calls `private_p`. Create a method `X_p` equal to `p`. Note that when the algorithm goes up in the `X` hierarchy, only one class is modified. When it goes down, all subclasses are changed. Note that `private_p` and `X_p` are just method names that do not appear anywhere in the program.

The semantics of the original C++-M code is preserved in the translated Oberon-M code because the renaming of methods neutralize any alien classes introduced between the class and its superclasses. Let us see the example of [Fig. 3]. The changes described above were applied to the hierarchy (a) producing the hierarchy (b). 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.

When we translate the hierarchy of [Fig. 3 (b)] to Oberon-M, we get `D-C-B-A`. In objects of `D`, the message send `this.C_p()` in method `C::m` will call 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.

Does this flattening of the hierarchy changes the meaning of message sends to variables ? That is, if `x.m()` calls at runtime a method `X::m` in the C++-M code, does this call a different method in the Oberon-M code ? There is a runtime search for a method for the message send "`x.m()`". The search for a method `m` begins at the class of the object `x` refers to at runtime. Suppose this class is `X`, which may inherit from several superclasses in a complex hierarchy in the C++-M code. However, there will never be an ambiguity on this search because either `X` define a `m` method or just one of the direct superclasses of `X` supplies one (which may be inherited from a superclass). Then it does not matter if this search is made in a multiple or in a single inheritance hierarchy. To understand that, suppose that, in C++-M, the search for a method `m` starts in class `X` and a method `m` is only found at a direct or indirect superclass `Y` of `X`. In Oberon-M, the search will start in class `X` also. However, some classes may have been introduced between `X` and `Y`, classes that are not between `X` and `Y` in the original C++-M hierarchy. It does not matter, because these inserted classes do not have a `m` method according to the definition of the C++-M model. Then method `m` will be found in class `Y` as in the C++-M code.

There is another problem with the example of [Fig. 3 (a)]. Suppose classes `A` and `B` define a method `t` (not shown in the Figure). A statement `super.t()` in class `C` will call method `A::t` at runtime. After the translation to Oberon-M, the hierarchy of this Figure 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 `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.

It is time to give the details of the translation of C++-M to Oberon-M, which is made below. 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].

A C++-M program is composed by classes which can be divided in several class hierarchies: $\mathtt{H}_C\mathtt{(A}_1\mathtt{)}$, $\mathtt{H}_C\mathtt{(A}_2\mathtt{)}$, ..., $\mathtt{H}_C\mathtt{(A}_n\mathtt{)}$. By the definition of $\mathtt{H}_C\mathtt{(A)}$, $\mathtt{H}_C\mathtt{(A}_i\mathtt{)} \cap \mathtt{H}_C\mathtt{(A}_j\mathtt{)} = \emptyset$ if $i \neq j$. The translation algorithm is then

for each hierarchy $\mathtt{H}_C\mathtt{(A)}$ of the C++-M program do:
    let S = $\mathtt{H}_C\mathtt{(A)}$

do a topological ordering of the set S obtaining B_n, ..., B_2, B_1
in which B_1 has no superclass and B_n has no subclass.
create a single inheritance hierarchy B_n-...-B_2-B_1 in Oberon-M.
for each class X of the set B_n, ..., B_2, B_1 do:
    for each message send this.p(...) of class X
    in which p is a public method, do:
        in the Oberon-M code, change this message send to this.X_p
        where X_p is a name that does not appear anywhere
        in the program.
        do a search for a method p starting in class X and
        continuing in the superclasses of X (in the C++-M hierarchy).
        suppose method p is found in class Y.
        rename this method to private_p (a new name) and move it
        to the private part.
        in the Oberon-M code, create public methods p and X_p.
        these methods just call private method p
        passing its real parameters to it.
        for each class Y ∈ allSubclassesOf(X) in C++-M, do:
            if class Y has a method p
            then
                in Oberon-M, rename method p to private_p
                (a new name) and move it to the private part.
                in Oberon-M, create public methods p and X_p.
                these methods just call private method p passing
                its real parameters to it.
            endif
    for each message send super.t(...) or super(W).t(...)
    in class X of C++-M, do:
        in C++-M, do a search for method t starting in W or in
        the superclass of X (if there is only one).
        the search continues in the superclass, superclass of the
        superclass, and so on. Let Y be the class in which
        method t is found.
        in class Y in Oberon-M, create two new methods:
        private_t and Y_t.
        the original method Y::t is renamed to private_t and moved
        to the private part of the class. Methods t and Y_t are created
        in the public part. These methods just call method private_t
        passing its real parameters to it.

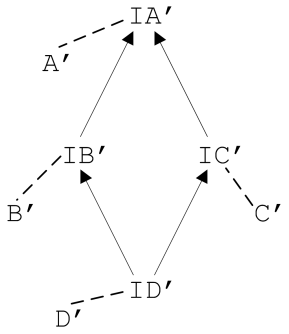

Figure 4: Interface hierarchy in Java-M that parallels that of the C++-M code

This algorithm orders the classes of the C++-M hierarchy $\mathtt{H}_C\mathtt{(A)}$ in topological order $\mathtt{B}_n$, ..., $\mathtt{B}_2$, $\mathtt{B}_1$ (of course, some $\mathtt{B}_i$ is `A`). The topological order preserves the subclassing 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 are not changed in the translation.

Java-M to C++-M

Translation from a language with single inheritance with interfaces to language with multiple inheritance.

For each interface `I` in Java-M, create an abstract class `I′` in C++-M. If `I` inherits interface `J` in Java-M, `I′` inherits class `J′` in the C++-M code.

For each class `A` in the Java-M code, create a class `A′` in the C++-M code with the same body. If `A` implements interface `I`, make `A′` inherit from `I′`. If `A` inherits from class `X`, make `A′` inherit from `X′`.

Whenever `X` is subtype of `Y` in the Java-M code, `X′` will be subtype of `Y′` in the C++-M code. Then the subtype relationships are preserved by this scheme. Since there is no other translation, the C++-M code will have the same behavior as the Java-M code.

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 hierar-

chies found in the original code. The second way is to flatten every class and its superclasses to a class without ancestors, as was made in the translation from Oberon-M to ClassMorph-M. Then hierarchies of interfaces recreate the hierarchies of the C++-M code. Let us detail this second way of translating C++-M to Java-M.

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. 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. 4].

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 mimics the class hierarchy of C++-M.

Consider an assignment `x = y` in C++-M in which the declared 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(...)` in C++-M is translated to `new A'(...)` in Java-M. Consider that:

- expression `new A(...)` in C++-M has type `A`. This expression can be used whenever an object of a supertype of `A` is expected;
- expression `new A'(...)` in Java-M has type `A'` and is subtype of `IA'` (it implements `IA'`).

If `A` is a subtype of `X` in the C++-M code, `IA'` is a subtype of `IX'` in Java-M. If the expression `new A(...)` is correctly used in C++, it is used where an object of `A` or a supertype (like `X`) is expected. After the translation, in Java-M, `new A'(...)` is used where `IA'` or a supertype (like `IX'`) is expected. Then no

compile-time type error is introduced by the translation.

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′`.

Java-M to Green-M

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

Each class `A` is translated to a class `A′` in Green-M with the same body and superclass, if any. Each interface is translated to an abstract class. Any other statement or declaration is the same in Java-M and in Green-M.

This scheme preserves subtyping. If `X` is supertype of `Y` in Java-M, then `Y` has at least the method signatures of `X`. This is true either because of the inheritances, in which methods are inherited, or because implementations of interfaces, in which the compiler demands that a class implements the interface methods. Therefore, `X` is a supertype of `Y` in Green-M. Note that `X`, `Y`, or both may be interfaces in Java-M and that both are classes in Green-M.

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 (in the real language Green, there is a class `Any`

that is inherited by any class that does not inherit from any other). 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 parameters 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 (much like class `Store` of [Fig. 1]). Every wrapper class has methods corresponding to the operations the basic type supports. For example, class `Int` has methods

```
Any plus (Any other )
Any minus(Any other )
Any mult (Any other )
Any div  (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. And "`x = 1`" is translated to "`x = new Int(1)`". This is type-

correct: `x` has type `Any`, supertype of `Int`.

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.

Smalltalk-M to Java-M

Translation from dynamically-typed language to language supporting single inheritance with interfaces.

This translation is almost equal to the translation from Smalltalk-M to Green-M. We will only describe the differences between the two. A class `A` in Smalltalk-M is translated to class $\texttt{A}'$ in Java-M. For each method `m` of `A`, create in Java-M an interface `Method_m_k` as in the Smalltalk-M to Java-M translation. Class $\texttt{A}'$ implements every interface `Method_m_k` corresponding to every method of `A`, including those inherited. In this way, it is correct to translate

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

to

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

At runtime, suppose `x` refer to an object of class `X` in Smalltalk-M. If this message send is legal, class `X` or any of its superclasses (direct or indirect) declares a method `m`. In Java-M, `x` will refer to an object of $\texttt{X}'$. Since `X` or any of its direct or indirect superclasses defines a method `m` with `k` parameters, then $\texttt{X}'$ implements interface `Method_m_k`. Therefore the translated message send is correct in Java-M too.

The rest of the translation is equal to the translation from Smalltalk-M to

Green-M.

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`, ...,` T_k x_k`)`

interface `Method_m_`T_1`_`T_2`_..._`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. That is, if a variable or return value of method has type `B` in Green-M, `B` a basic type, it should have the same type in the translated Java-M code. Every literal or expression in Green-M, such as `0`, `true`, or `1 + 2` is translated to itself 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′`. This 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 in one language is equal to the other, the method called at runtime will be the same, `X::m`.

3 Conclusion

[Fig. 5] shows all translations described in the previous section. An arrow from `L` to `M` means code in model `L` can be translated to code in model `M` by a scheme described in this article. Since there is a path starting in any vertex and ending in any other, code in any of the six models employed in this paper can be translated to any other. This is a little bit surprising. Some models are clearly more polymorphic than others and it seems very difficult to do some translations. Let us study these.

1. The type system of Green offers more polymorphism than that of Java. In Green, assignment "`a = b`" is legal if the declared type of `b`, say `B`, has at least all the public methods of the declared type of `a`, say `A`. It does not matter whether `B` inherits from `A` or not. In Java, all subtype relations have to be *explicitly* declared by the programmer. We declare that `Y` is a subtype of `X` if `Y` inherits from `X` (directly or indirectly), `Y` implements the interface

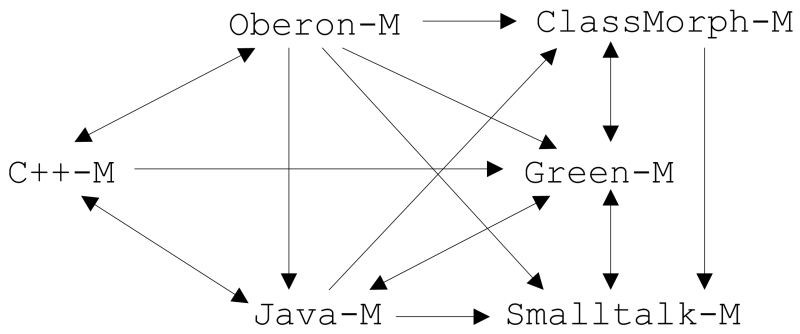

Figure 5: Graph showing translations between the language models

`X` (directly or indirectly), or `Y` is subtype of a subtype of `X`. It seems difficult to go from a free type system as that of Green to a more strict like that of Java. However, we can translate Green code to Java, albeit in a complex way.

2. Multiple inheritance offers the possibility of 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 to 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.

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 compiler and its source code are freely available.

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:

– when translating Oberon-M to ClassMorph-M, a class and its superclasses are flattened into a single class. The consequences of this are that message sends to `this` in Oberon-M, which demand runtime searches, become message sends to `this` in ClassMorph-M, which are linked to the method at compile-time. In Oberon-M, there is a runtime search in message sends to

`this`, a time-consuming operation. In ClassMorph-M, there is none. This is not for free. Every superclass method must be duplicated in every subclass because the method to be called in a message send to `this` is different in every subclass. This translation scheme can be employed, with restrictions, from Oberon-M to Oberon-M. The restriction is that some subtypes in the source code may not be subtypes in the target code. Therefore, we can exchange code for speed in some occasions by changing message sends to `this` as described by the translation scheme Oberon-M—ClassMorph-M. Note that we can eliminate polymorphism in message sends to `this`, in this case, but we cannot eliminate it in message sends to variables. It is impossible to design an algorithm that finds exactly which methods will be called at runtime by a given message send. This is uncomputable;

- basic types and values in Oberon-M, Java-M, Green-M, and ClassMorph-M can be translated to basic types of Smalltalk without modifications. Everything related to basic types that is allowed in the source languages is also allowed in Smalltalk-M;

- 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. And ClassMorph-M is a subset of Green-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 language to Smalltalk-M because this language 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 languages;

- 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;

- Java-M can be translated to ClassMorph-M using the scheme Oberon-M to ClassMorph-M. This is possible because the subtype definition of ClassMorph-M encompasses that of Java-M, even though ClassMorph-M does not support inheritance and Oberon-M does not support multiple subtyping;

- C++-M can be translated to the single inheritance language Oberon-M. However, this is only possible by wholesale creation of methods to separate sister classes in the flattened class hierarchy of Oberon-M. Of course, a lot of optimizations can be made in the translation, which were not described;

- 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. 4], 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 superclassless Green-M class 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 all the 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

  `((Method_m_k ) x).m(`x_1`, `x_2`, ..., `x_k`)`

- the translation from Green-M to Java-M shows how different are the subtype definitions in the 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. Probably it is 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.

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

[Fowler et al. 1999] Fowler, M.; Beck k.; Brant, Opdyke, W.; Roberts, D.: "Refactoring: Improving the Design of Existing Code"; Addison-Wesley, 1999.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software"; Addison-Wesley, Reading, MA, 1994.

[Goldberg and Robson 1983] Goldberg, A., Robson, D.: "Smalltalk-80: The Language and its Implementation"; Addison-Wesley, Reading, MA, 1983.

[Gosling et al. 2007] Gosling, J., Joy, B., Steele, G., Bracha, G.: "The Java Language Specification"; third edition, Prentice Hall PTR, 2005. Available at http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

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

[Guimarães 2007] Guimarães, J.: "The Green language"; Available at http://www.dc.ufscar.br/~jose/green/green.htm, 2007.

[Guimarães 2006] Guimarães, J.: "The Green language"; Computer Languages, Systems, & Structures, 32, 4, 2006, 203-215.
[Guimarães 2003] Guimarães, J.: "Experiences in Building a Compiler for an Object-Oriented Language"; SIGPLAN Notices, 38, 4, 2003, 25-33.
[Niklaus 1988] Wirth, N.: "The Programming Language Oberon"; Software, Practice & Experience 18, 7, 1988, 671-690.
[Stroustrup 1991] Stroustrup, B.: "The C++ Programming Language"; third edition, Addison Wesley, Reading, MA, 1991.