Conception of a static oriented language: an overview of SCOOL

Thomas Moulard

revision 963

SCOOL is a static oriented language designed to solve the problems encountered during the development of OLENA. Despite of the advantage brought by Metalic and Static, adding new classes remains a difficult work for a user who isn’t used to deal with the internal mechanisms of OLENA. SCOOL allows fast application prototyping via a simple syntax which is more expressive than C++ syntax.

Keywords
language, compilation, object oriented, concept, where clause, olena

Laboratoire de Recherche et Développement de l’Épita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
thomas.moulard@lrde.epita.fr – http://www.lrde.epita.fr/
Copying this document

Copyright © 2006 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.
Contents

1 Introduction 4

2 Why a static object language? 5
  2.1 C++ limits 5
     2.1.1 Constraint on template 5
     2.1.2 Multi-method 6
     2.1.3 Covariant method call 6
  2.2 SCOOL’s characteristics 7
     2.2.1 SCOOL’s features 7
     2.2.2 Benchmarking static typed programs 7
  2.3 A (not so) simple example 8

3 Our tools 11
  3.1 Metalic 11
  3.2 Static 12
     3.2.1 Overview of Static 12
     3.2.2 SCOOP 12

4 SCOOL’s syntax 14
  4.1 Overview of the syntax 14
  4.2 Where clause 14
     4.2.1 Verification of a function’s parameter 14
     4.2.2 Static overloading 14
  4.3 Multi-method 16
     4.3.1 The “classic” mechanism 16
     4.3.2 Another use of where clauses 17
  4.4 Virtual type definition 17

5 Conclusion 19

6 Bibliography 20
Chapter 1

Introduction

This technical report presents a language named SCOOL for “Static C++ Object Oriented Language”. We will focus on two points: why a static language is a good idea and how to express a program in SCOOL.

Our goal is designing a tool that allows programmers to write fast prototyping. For instance it’s useful for OLENA because designing a new class hierarchy is a difficult work. Static inheritance and the numerous templates makes this work uncomfortable for users that doesn’t have to understand the internal mechanisms of OLENA. On top of that, SCOOL allows us to express constructions that should be really difficult to write in C++.

Even if this project was launched by the OLENA team, this tool can be useful for anyone which is interested by the generation of fast C++ code via metaprogramming and static typing.
Chapter 2

Why a static object language?

First, we will try to see why we need a new language to express our programs. SCOOL’s features will also be detailed to highlight the difference between the complex C++ code and our syntax that will be introduced at the chapter 3.

2.1 C++ limits

2.1.1 Constraint on template

The first huge problem of C++ is linked to templates. When we write a function with a parameter, we often want to restrict the set of parameters that the function should accept.

Program 2.1 An illustration of constraint on a template

```c++
1 template <typename T>
2 void foo(T& x)
3 {
4   x.foo();
5 }
6
7 struct MyClass {
8   void foo() {}
9   }
10
11 int main()
12 {
13   MyClass mc;
14   foo(mc);
15   foo(2);
16 }
```

In this example, a templated function is called twice. The first call at line 14 is correct because MyClass has a method called foo. But the call at line 15 isn’t valid because in this case T is an integer.

By consequence, when this source will be compiled, the compiler will fail on line 4 and notices that “foo” doesn’t exit in x. A better error should be “On line 15: incorrect type of T”.

To manage this issue, we need concepts (D. Gregor, 2000, Concepts for C++0x). With this mechanism, type checking can be made on templates.

2.1.2 Multi-method

The multi-method is the second mechanism that really miss to C++.

A multi-method is a method that will be dispatched depending on the exact type of its arguments.

To illustrate this definition, here is an example of multi-method.

Program 2.2 An example of multi-method in pseudo-C++

```cpp
void assign (Image& i)
{
    // Default case
}

void assign (Image1d& i)
{
    // Implementation for 1d image
}

void assign (Image2d& i)
{
    // Implementation for 2d image
}

void main()
{
    Image a = new Image1d();
    Image b = new Image2d();
    Image c = new Image3d();

    assign(a); // assign(Image1d&) is called.
    assign(b); // assign(Image2d&) is called.
    assign(c); // assign(Image&) is called.
}
```

This example is not a C++ program. If this example were compiled, `assign(Image)` will be called three time.

With multi-methods, this program does exactly what we expects.

2.1.3 Covariant method call

This is a well-known problem that can be easily described by an example:

We would like to write such programs but as method signatures are invariant in C++, we can’t.
**Program 2.3** Example of covariant call

```c++
struct A {
};

struct A_prime : public A {
};

struct B {
    void foo(A&);
};

struct B_prime : public B {
    void foo(A_prime&);  // there is no overloading here:
                          // as signature are different from B::foo, a
                          // new method is defined!
};
```

### 2.2 SCOOL’s characteristics

#### 2.2.1 SCOOL’s features

SCOOL is a language that is transformed to C++ and then compiled. The Stratego toolkit is used to perform program’s transformations. Moreover, SCOOL is designed for static typing. In this language, the exact type of a class is known at compile time and everywhere in the source code. The aim is producing fast C++ programs by avoiding runtime verifications.

#### 2.2.2 Benchmarking static typed programs

To illustrate the fact that static typing improves execution’s speed, here is a benchmark of the assignment of a color to all the points of an image:

<table>
<thead>
<tr>
<th></th>
<th>Dedicated</th>
<th>Static</th>
<th>Object oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>Java</td>
<td>C++</td>
</tr>
<tr>
<td>ref.</td>
<td>0.0062 s</td>
<td>0.0062 s</td>
<td>0.1132 s</td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL-like</td>
<td>0.0071 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOOP</td>
<td>0.0062 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Object oriented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>0.1091 s</td>
<td></td>
<td>1.091 s</td>
</tr>
<tr>
<td>Eiffel</td>
<td>0.1091 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>0.0240 s</td>
<td>3.9 when 1 run</td>
<td>0.0143 s</td>
</tr>
<tr>
<td>Java</td>
<td>0.0143 s</td>
<td>2.3 when 10 runs</td>
<td>0.0129 s</td>
</tr>
</tbody>
</table>

Table 2.1: SCOOL uses SCOOP to produce fast C++.
The major fact that should be noticed are:

- There is three types of programs: dedicated code (no abstraction), object oriented code and static code.
- Of course, dedicated code is really fast but there is no genericity.
- With usual programming paradigm, there is a huge factor between dedicated code and object oriented code. One reason is the dynamic function's dispatch (i.e. the virtual keyword in C++).
- To conclude, static typing with SCOOP allows genericity at no cost!

The remaining question is what exactly are the advantage of SCOOL? We will see this via a simple example.

2.3 A (not so) simple example

The aim of this example is building a simple image hierarchy. The following UML diagram shows the detail of our hierarchy.

Figure 2.1 A simple image hierarchy

As you can see in the program 2.4, this example is quite complex, despite of the fact that we just wanted to implement a very simple hierarchy. In this context, you can easily understand why a simplified language like SCOOL is required to hide the complex C++ translation.

For instance, final user shouldn’t see static mechanisms like traits. A trait is a static function that takes a parameter in entry and computes a result that may be a type.

To compare, the program 2.5 is the SCOOL equivalent of the previous example.

This sample is shorter and easier to understand than the last. The syntax is natural and can be written by anyone. This is the aim of SCOOL: allowing final user to add their own classes easily, even if the C++ translation is difficult to understand.

However, we’ll see what mechanisms are used to produce the generated C++ source code. That’s why i’ll now described the tools used by SCOOL.
Why a static object language?

**Program 2.4** C++ version of a sample image hierarchy

```cpp
template<typename T>
struct traits {};

template<typename E>
struct traits <Image<E> > {
    typedef undefined iterator_vt;
    typedef undefined value_vt;
};

template<typename Data>
struct traits <Image1d<Data> >:
    public traits <Image<Image1d<Data> > > {
    typedef Iterator1d<Data> iterator_vt;
    typedef Data value_vt;
};

template<typename E>
struct Image : public Any<E> {
    vtype(E, iterator) first();
    vtype(E, iterator) last();
};

template<typename Data>
class Image1d : public Image<Image1d<Data> > {
    public:
        Image1d (unsigned l);
        Iterator1d<Data> impl_first();
        Iterator1d<Data> impl_last();
    private:
        unsigned l_;
        Data* d_; 
};
```
Program 2.5 SCOOL version of a sample image hierarchy

```c
1 Image : class = { 
2     public { 
3         decl iterator : type; 
4         decl value : type; 
5     } 
6 } 
7 Image1d : [Data : type] -> class < Image => 
8     public { 
9         iterator : type = Iterator1d[Data]; 
10        value : Data; 
11        first : () -> iterator = {} 
12        last : () -> iterator = {} 
13     } 
14     private { 
15         l_ : unsigned; 
16         d_ : buffer[Data]; 
17     } 
18 }
```
Chapter 3

Our tools

3.1 Metallic

Metalic is metaprogramming library used to make easier the writing of static equipment for C++ programs.

For example, Metallic provides build-in support of:

- static if
- static switch...case
- static assertions
- ...

To illustrate this, here is a small sample:

Program 3.1 An illustration of constraint on a template

```cpp
struct alpha;
struct beta;
struct gamma;
struct delta;

int main()
{
    typedef mlc_if_(mlc::bexpr_<true>, alpha, beta) x;
    mlc::assert_<mlc_eq(x, alpha)>::check();

    typedef mlc_if_(mlc::bexpr_<false>, gamma, delta) y;
    mlc::assert_<mlc_eq(y, delta)>::check();
}
```

This program declares four structures and uses two static if and two static assertions.

The first “if” at line 9 produces an instance of alpha if the condition is valued to true, otherwise an instance of beta is created. As the condition is true, x is an instance of alpha. The line 10 checks if x is an instance of alpha or not. If the condition fails, the program won’t compile.
The second part of the program follows the same scheme with a false condition. After seeing the Metalic library, it’s time to discover Static and the SCOOP paradigm.

3.2 Static

3.2.1 Overview of Static

Static is a library that implements the SCOOP paradigm by using the Metalic library. The aim of SCOOP is providing static inheritance and by consequence static typing for C++ programs. The main mechanisms of Static are:

- Static inheritance.
- Multi-method.
- Virtual types.

3.2.2 SCOOP

SCOOP is a paradigm designed to produce fast programs by avoiding runtime checking. In other words, the exact type of each class is a parameter of the class. So the exact type is known, everywhere, at no cost.

Figure 3.1 Example of static inheritance

In the program 3.2, we see that the virtual keyword of C++ is simulated through a system of method’s delegation. We can see that algorithm retrieves the exact type of the object through the method, then the implementation is called.

The method exact is simple: as the exact type of each class is a parameter, exact is just a cast to the exact type. To equip all classes with this method, we use a unique super-class called Any.

Finally, we achieve to simulate the virtual keyword of C++ with no cost with SCOOP. Furthermore, this paradigm has other features like multi-methods: as we always known our exact type, it’s really easy to dispatch a method depending on the exact type of each parameter. All these features will be useful to translate SCOOL into C++.
Program 3.2 An illustration of constraint on a template

```cpp
1 template<typename E>
2 class Any
3 {
4   public:
5     E& exact () { return *(E*)(void*)this; }
6     const E& exact () const { return *(E*)(void*)this; }
7   }
8
9 template<typename E>
10 struct Image: public Any<E>
11 {
12   void algorithm ()
13   {
14     this.exact().impl_algorithm();
15   }
16 }
17
18 struct Image1d: public Image<Image1d>
19 {
20   void impl_algorithm () {}
21 }
22
23 struct Image2d: public Image<Image2d>
24 {
25   void impl_algorithm () {}
26 }
```
Chapter 4

SCOOL’s syntax

4.1 Overview of the syntax

After seeing the reason behind the SCOOL project and the tools we use, it’s time to see the
SCOOL’s syntax.

To begin, comments are written in the same way than in C++.

The syntax “identifier : type” is always used in this language because it is easier to read than
the C++ syntax.

Here is an overview of the syntax of our language:

A main characteristic of SCOOL is the meaning of the brackets: the brackets indicates a static
parameter or a source code which is evaluated at the compilation. For example, the $T$ parameter
of the Image1d class will be translated into a template in C++.

The language obviously supports class declaration with a syntax near to the C++. The additional
qualifier final in a class declaration allows to declare a final class, ie. a class from which
no inheritance is possible.

The inheritance uses the symbol >. In a declaration if \texttt{Beta : class < Alpha} is written
then Beta’s mother class is Alpha.

The attribute declaration is really near the C++’s syntax but there is some special variables
which are defined in our last example: \texttt{coord_t}, \texttt{point_t} and \texttt{value_t} are virtual types.
They are unknown types in the upper classes, their real values are defined in lower classes. The
virtual types will be explained in the last part of this report.

4.2 Where clause

4.2.1 Verification of a function’s parameter

A where clause is a static verification on a static parameter of a function. It allows to make
checking you can’t write easily in C++ with templates.

In this example, \texttt{T} must be a subclass of \texttt{A}. If this is not the case, an error will occur at the
compilation.

4.2.2 Static overloading

Another use of where clause exists: if two methods with the same signature are written in a
class, the where clause can be used to create a new kind of overloading.

This program shows a possibility of overloading with where clause.
Program 4.1 A tour of the SCOOL’s syntax.

```java
/* Abstract classes */
Point : class = {
    public {
        decl coord_t : type;
    }
}

Image : class = {
    public {
        decl value_t : type;
        decl point_t : type;
        decl get : (p : point_t) -> value_t;
    }
}

/* Concrete classes */
final Point1d : class < Point = {
    public {
        coord_t : type = uint16;
        l : coord_t;
    }
};
final Image1d : [T: type] -> class < Image => {
    public {
        value_t : type = T;
        point_t : type = Point1d;
        get : (p : point_t) -> value_t => data.at(p.l);
    }
    private {
        data : vector[value_t];
    }
}
```

Program 4.2 A method with a constraint on a parameter

```java
A : class;
foo : [T : type where T < A] -> void =
{
    // ...
}
```

To begin, the keyword `overloading` is new: it indicates that many implementations of this method will be written and where clause overloading algorithm will be used to dispatch the calls.

We can also notice that a third implementation with no where clause could be written to produce a default case for the overloading. If no default case are written, the compilation will fail if no case matches with a call of this method.
4.3 Multi-method

A multi-method is a method which is dispatch depending on the type of its arguments. Multi-methods are not supported by C++ but the SCOOP paradigm can simulate a multi-method as the exact type is always known.

4.3.1 The “classic” mechanism

The first mechanism for multi-method in SCOOL uses the keyword multimethod. It indicates that the method which is defined will be a multi-method, so the dispatching policy will be the multi-method’s policy.

Program 4.4 A "classic" mechanism of multi-method

```plaintext
1 multimethod clear :
2 (img : ref image) -> void;
3
4 impl clear :
5 (image : ref image1d) -> void =
6 |
7 // ...
8 |
9 impl clear :
10 (image : ref image2d) -> void =
11 |
12 // ...
13 |
```

The two implementations illustrates how to define a multi-method. It looks like C++ overloading, the only difference is the method’s definition with the multimethod and impl keywords.

The first line is the definition of the multi-method with the keyword multimethod. Then, the implementations are written, they must contain the qualifier impl.

If a subclass of Image1d or an instance of Image1d is used as an argument of the function clear, the first implementation will be called. In the same way, if a subclass of Image2d or an instance of Image2d is used as an argument of the function clear, the second implementation will be called.
The unique super class Any can be used to create a default case. If no default case exists, calls that don’t match will fail at the compilation.

4.3.2 Another use of where clauses

A multi-method can also be declared though a where clause.

Program 4.5 Another way of expressing the previous example.

```cpp
1 overloading clear : [T : type where T < image1d] (image : ref T) -> void =
2 |
3 // ... |
4 |
5 overloading clear : [T : type where T < image2d] (image : ref T) -> void =
6 |
7 // ... |
8 |
9 }
```

We can see that this example has exactly the same behavior than the example given at the previous section. So, we can conclude that where clauses are a strongest mechanism than multi-methods. In spite of that, the multimethod keyword is useful to define a multi-method easily.

4.4 Virtual type definition

The last feature of SCOOL is the virtual type. A virtual type is a type which can be undefined in a class and refined in a subclass.

This is a really strong feature that is implemented in C++ through a set of traits. A trait is a meta-programmation function that takes types as parameters and returns a type.

This feature allows to realize method’s covariance, this fact will be illustrate by the program 4.6.

Figure 4.1 A famous problem...

We have here a complete hierarchy with a covariance method: eat can take as argument a Banana if the current object is an instance of Ape. At the opposite, if the current object is an instance of Cow, the argument of eat will be an instance of Grass. This can’t be realized easily in C++ because arguments are a part of the signature and if the signature changes, overloading is not possible. With the vtype system, the signature doesn’t change and overloading is now possible.
Program 4.6 An illustration of a covariant call

```java
Food : class = {}

Animal : class =
public {
  decl food : type;
  decl eat : (f : food) -> void;
}

Grass : class < Food = {}

Cow : class < Animal =
public {
  food : type = Grass;
  eat : (f : food) -> void =
}

Banana : class < Food = {}

Ape : class < Animal =
public {
  food : type = Banana;
  eat : (f : food) -> void =
}

main : () -> int =
{
  var Ape a;
  var Cow c;
  var Banana b;
  a.eat(b);
  c.eat(b); // This is invalid.
}
```
Chapter 5

Conclusion

To conclude this report, we have made a lot of progresses and the foundation of the language is ready.

- The grammar’s specification is finished (except for some low priority features).
- Transformation of samples from SCOOL into C++ has been made.
- The implementation of the compiler has been started and some complete programs can be compiled, even if all features are not currently functional.

Obviously, some work is still required to finish the implementation of the compiler, we can quote:

- We need to finish the implementation of some high priority features,
- then we could manage to add low priority mechanisms,
- And our final goal is using SCOOL in OLENA and the others projects of the LRDE.
Chapter 6

Bibliography