SCOOL: Generic programming and concepts

Maxime van Noppen

revision 1746

SCOOL is a static object-oriented language. It has been created to help one to take advantage of all the power of static C++ thanks to a more expressive syntax. It is not directly compiled but is translated into C++. This year was quite important for the project. Indeed, there are tight links between SCOOL’s development and the generic image processing library MILENA from the OLENA platform. As some big changes occurred in the library, work needs to be done to adapt the language to its new paradigms and to its new needs. Work also needs to be done to continue the implementation of the different features of SCOOL. This year the work will be focused on concept-oriented programming. This allows one to easily express constraints on generic programming.

SCOOL est un langage statique orienté objet qui a été créé afin de pouvoir utiliser toute la puissance du C++ statique de manière plus aisé grâce à une syntaxe plus expressive et agréable. Il n’a pas pour but d’être directement compilé mais d’être traduit en C++. Cette année le travail revêt une importance particulière. En effet, SCOOL est développé en étroite collaboration avec l’équipe de développement de la bibliothèque de traitement d’image MILENA de la plate-forme OLENA ; l’an passé a été pour elle le cadre de grands changements internes. Un des axes majeurs du développement de SCOOL va donc être de s’adapter aux nouveaux paradigmes et aux nouveaux besoins de la bibliothèque. Le second axe essentiel de travail est la poursuite du développement du langage. Cette année le travail va être concentré sur la programmation par concepts qui est une approche permettant de formaliser facilement des contraintes sur la programmation générique.

Keywords
C++, Transformers, program transformation, SCOOL, OLENA, STATIC, METALLIC
Copyright © 2008 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.

Copying this document
Contents

Introduction 5

1 State of the Art 6
  1.1 THE PIVOT ........................................... 6
  1.2 METAGENE ........................................... 7
    1.2.1 Generalities ................................... 7
    1.2.2 Metaprogrammation .............................. 8
  1.3 D .................................................... 9
    1.3.1 Generalities ................................... 9
    1.3.2 Traits .......................................... 9
    1.3.3 Metaprogrammation .............................. 10
  1.4 NEMERLE ........................................... 12
    1.4.1 Generalities ................................... 12
    1.4.2 Metaprogramming ................................ 12

2 Getting in touch with SCOOL 14
  2.1 Syntax basis ....................................... 14
  2.2 Verbatim C++ ....................................... 15

3 Types 16
  3.1 Constness and references ........................... 16
  3.2 Type contraints: where clauses, .................. 17

4 Functions 18
  4.1 Generalities ....................................... 18
  4.2 Dynamic functions .................................. 18
  4.3 Static functions .................................... 19
    4.3.1 Declaration .................................... 19
    4.3.2 Static if statement ............................. 20
    4.3.3 Returning pre-existing type ..................... 20
    4.3.4 Creating a type on the fly ..................... 21

5 Classes 23
  5.1 Generalities ....................................... 23
  5.2 Declaration ........................................ 23
  5.3 Access blocks ...................................... 23
  5.4 Methods and variables ............................. 24
  5.5 Constructor and destructor ........................ 24
## CONTENTS

5.6 Inheritance ................................................. 25  
5.7 Parameters ................................................ 26  

6 Concepts .................................................. 27  
6.1 Generalities ............................................... 27  
6.2 Basics .................................................... 27  
6.3 C++ translation .......................................... 28  

7 Examples and translations ................................. 30  
7.1 Introduction .............................................. 30  
7.2 Simple hierarchy ....................................... 30  

8 Future work ............................................... 32  
8.1 Classes .................................................. 32  
8.1.1 Object construction ................................. 32  
8.2 Concepts ............................................... 33  
8.2.1 Multiple concept modeling ......................... 33  
8.2.2 Static SCOOL check algorithm ..................... 33  
8.3 Type checking ........................................... 33  
8.4 Static statements ....................................... 33  
8.5 Where clauses - static dispatcher ...................... 34  
8.6 Functions ................................................. 34  
8.7 Grammar and translator improvements ............... 34  

Conclusion .................................................. 35  

Bibliography ................................................ 36  

A Static dispatcher ......................................... 37
Introduction

SCOOL, which stands for Static Object Oriented Language, has been created in 2006 (Moulard, 2007b) to address a big C++ issue: its lack of powerful high-level static constructions. Indeed, thanks to its template system, C++ allows one to perform very advanced static computations at compile time. However, the underlying code is often complicated, large and difficult to maintain. SCOOL aims at providing easy to use high-level static constructions to perform those computations. Therefore, it is not directly compiled to machine code but is rather translated into C++ equipped with meta-algorithms that come from the OLENA project ((Géraud, 2006), (Géraud and Levillain, 2008)). Quentin Hocquet started the implementation of SCOOL (Hocquet, 2006) focusing on classes and static inheritance.

This report is a general documentation of SCOOL, presenting all its features. Though, the important parts that were worked on this year are the concepts (see chapter 6 on page 27) and static functions (see section 4.3 on page 19).

The core of this report is focused on SCOOL’s features and not on its translation in C++. They are present only when it is important to see how the SCOOL code is translated to understand its behaviour. However there is a dedicated chapter Examples and translations (chapter 7 on page 30) that provides examples with their translation into C++.

Acknowledgments

- Akim Demaille, Thierry Géraud, Roland Levillain: Design of SCOOL and C++ translations
- Valentin David, Nicolas Pierron: Help on STRATEGOXT.
- Roland Levillain, Florian Lesaint: Reviewer
- Laurent Le Brun: Help on the State of the Art (NEMERLE language)
- Ugo Jardonnet, Alexandre Fournier: Help on the State of the Art (D language)
Chapter 1

State of the Art

1.1 The Pivot

“The Pivot is a framework for static analysis and transformation of C++ programs, being developed at Texas A&M University. It aims at the support for high-level parallel and distributed programming techniques. It “understands” the higher levels of C++ (e.g. templates, specializations, concepts), crucial for advanced optimizations, validation of safety, enforcement of dialects, support for staging libraries.”

http://parasol.tamu.edu/pivot/

What is interesting in The Pivot (Stroustrup, 2004) from a SCOOL point of view is the XPR language. XPR is a high-level language that aims C++ program representation and which has a syntax close to SCOOL’s. It has several interesting ideals, some of them being common with ours:
• Be complete: represent all standard C++ constructs
• Be general: not targeted to a small area of applications
• Be regular: Try to have as general rules as possible rather than special cases
• Emphasize types
• Be compiler neutral
• Be efficient an elegant

As SCOOL is specially dedicated to the static world, the first rule will not apply: there will be C++ constructs that are not represented, like classic inheritance (see section 5.6 on page 25). On the contrary, the other points are quite general and match SCOOL’s objectives.

Unfortunately, THE PIVOT website is still poor at this time and shows neither concrete examples nor usages of XPR so that we could have compared both languages.

1.2 METAGENE

1.2.1 Generalities

"The C++ language offers a two layer evaluation model. Thus, it is possible to evaluate a program in two steps: the so-called static and dynamic evaluations. Static evaluation is used for reducing the amount of work done at execution-time. Programs executed statically, called meta-programs, are written in C++ through an intensive use of class templates. Due to the complexity of these structures, writing, debugging and maintaining C++ meta-programs is a difficult task. Metagene is a program transformation tool which simplifies the development of such programs. Due to the similarities between C++ meta-programming and functional programming, the input language of Metagene is an ML language. Given a functional input program, Metagene outputs the corresponding C++ meta-program expressed using class templates."

http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/MetaGene

Though the METAGENE (Maes, 2004) project is no more maintained, a lot of interesting work has been done and some of the ideas developed have been implemented in SCOOL. For example, seeing parameterized classes like static functions (see section 4.3 on page 19) is a very interesting and powerful SCOOL feature that was first introduced in METAGENE.

The major difference with SCOOL, besides the language syntax, is that it is a language manipulating C++ code rather than being a representation of it. You could have found functions in METAGENE like:

```
let get_methods : cxxclass -> cxxprim list
let add_method : cxxclass -> cxxprim -> cxxclass
```

SCOOL aims to be a direct representation of C++ code though it is not only a neater syntax but a more powerful and expressive language giving easily access to features which require bunches of C++ code.
1.2.2 Metaprogrammation

The power function

Consider the *power* function. It is very natural to write it in METAGENE:

```plaintext
let rec pow x n =
    match n with
    0   -> 1
    1   -> x
    _   -> x * pow x (n - 1)
```

Once translated into C++, the function is represented by parameterized classes:

```plaintext
template <int x>
struct pow
{
    template<int n>
    struct res
    {
        enum { res = x * pow<x, n - 1> :: res };
    };

    template<> struct res<1>
    {
        enum { res = x };
    };

    template<> struct res<0>
    {
        enum { res = 1 };
    }
};
```

Static functions

Static functions (see section 4.3 on page 19) are functions taking only static parameters and returning a type. Though they haven’t been implemented in METAGENE they have been discussed and would have looked like:

```plaintext
let vector T = <c@ /* no base class */ @
    public:
        T& operator [] (unsigned i)
        { return _data[i]; }
    private:
        T* _data;
    @>
```
The introduction of the concept of static functions is very powerful. Indeed, you benefit of all the usual power of functions (partial or lazy evaluation, overloading, …) with a natural and easy syntax.

1.3 D

1.3.1 Generalities

"D is a systems programming language. Its focus is on combining the power and high performance of C and C++ with the programmer productivity of modern languages like Ruby and Python. Special attention is given to the needs of quality assurance, documentation, management, portability and reliability.

The D language is statically typed and compiles directly to machine code. It’s multiparadigm, supporting many programming styles: imperative, object oriented, and metaprogramming. It’s a member of the C syntax family, and its appearance is very similar to that of C++.”

http://www.digitalmars.com/d/

1.3.2 Traits

Compared to C++, the D language provides some interesting means to gather informations on types at compile-time. For example, the traits system. They allow one to access to informations such as:

- isAbstractClass
- isVirtualFunction
- isAbstractFunction
- hasMember
- getMember
- ...

For example (taken from http://www.digitalmars.com/d/2.0/traits.html):

```d
import std.stdio;

abstract class C { int foo(); }

void main()
{
    C c;
    writeln(_traits(isAbstractClass, C));
    writeln(_traits(isAbstractClass, c, C));
    writeln(_traits(isAbstractClass));
    writeln(_traits(isAbstractClass, int *));
}
```
Though this system is quite powerful, it is not extensible and a user cannot define its own traits.

### 1.3.3 Metaprogrammation

As in C++, metaprogramming in D is performed thanks to heavy template usage. Though, in C++ the discover of these techniques is more an accident than a formalized ability. This implies some weird behaviours and very complicated code. The D language has been designed keeping in mind those techniques to try to have them properly defined in the language.

"Templates in C++ have evolved from little more than token substitution into a programming language in itself. Many useful aspects of C++ templates have been discovered rather than designed. A side effect of this is that C++ templates are often criticized for having an awkward syntax, many arcane rules, and being very difficult to implement properly. What might templates look like if one takes a step back, looks at what templates can do and what uses they are put to, and redesign them?"


### Compile-time functions

The D language provides the ability to execute usual functions at compile time if it is possible. Obviously there are a lot of constraints on the function so that it can be executed at compile time. Here is a non-exhaustive list of some of them:

- Function arguments must all be:
  - integer, floating, character, string or array literals
  - struct literals where the members are all items in this list
  - const variables initialized with a member of this list
- the function may not be a non-static member, i.e. it may not have a this pointer
- expressions in the function may not:
  - throw exceptions
  - use pointers, delegates, non-const arrays, or classes
  - reference any global state or variables
  - reference any local static variables
  - new or delete
  - call any function that is not executable at compile time
- ...

Here is an example which enlightens this ability. Consider the `power` function that is either executed at compile time or at run time depending on its call the context.
```c
int power(int x, int n) // Computes x^n
{
    if (n == 0)
        return 1;
    if (n == 1)
        return x;
    return x * power(x, n - 1);
}

void main()
{
    static int res_1 = power(42, 3); // Computed at compile-time
    int res_2 = fibo(42, 3); // Computed at run-time
}
```

**Static if**

The D language provides the ability to execute static-ifs. This means that they will be computed at compile-time and that they can perform tests on types. The idea is very similar to what is implemented in SCOOL (see subsection 4.3.2 on page 20).

Here is an example illustrating the use of the static if statement and the template specialization.

```c
class Container(T)
{
    static if (is(T == int))
    {
        public int i_value;
    }
    else
    {
        public T t_value;
    }
}

class Container(T: double)
{
    public double d_value;
}

void main()
{
    auto a = new Container!(int);
    a.i_value = 42;
```


```csharp
auto b = new Container<double>;
b.d_value = 42;

auto c = new Container<char>;
c.t_value = 42;
```

Combining the power of the static `if` and of the traits allows one to write very powerful code very easily.

### 1.4 NEMERLE

"Nemerle is a high-level statically-typed programming language for the .NET platform. It offers functional, object-oriented and imperative features. It has a simple C#-like syntax and a powerful metaprogramming system.

Features that come from the functional land are variants, pattern matching, type inference and parameter polymorphism (aka generics). The meta-programming system allows great compiler extensibility, embedding domain specific languages, partial evaluation and aspect-oriented programming."

[http://nemerle.org/What_is_Nemerle](http://nemerle.org/What_is_Nemerle)

#### 1.4.1 Generalities

NEMERLE provides a powerful meta-programming system thanks to advanced *macros*. The idea is the same as the usual C macros except that they are much more powerful: they are statically type-checked, can contain static statements, or even extend the syntax of the language.

#### 1.4.2 Metaprogramming

**Static if**

Here is an example (taken from [http://nemerle.org/Macros_tutorial](http://nemerle.org/Macros_tutorial)) showing the static `if` in action. Everything is resolved at compile time.

```csharp
using Nemerle.Compiler.Parsable;

module MyModule
{
    public mutable debug_on : bool;
    public compute_some_expression() : PExpr
    {
        if (debug_on)
            <[ System.Console.WriteLine("Hello, I'm debug message") ]>
        else
            <[ () ]>
    }
}
```
Extending the syntax

Here is an example (taken from http://nemerle.org/Macros_tutorial) showing how to add a C-like for loop:

<table>
<thead>
<tr>
<th>Macro for (init, cond, change, body)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax (&quot;for&quot;, &quot;(&quot; init &quot;;&quot;, cond &quot;;&quot;, change &quot;)&quot;, body)</td>
</tr>
</tbody>
</table>
Chapter 2

Getting in touch with SCOOL

2.1 Syntax basis

SCOOL is based on a non-ambiguous and consistent syntax. All the declarations respect the following syntax:

\[
\begin{align*}
  \text{identifier} & : \text{type}; \\
  \text{identifier} & : \text{type} = \text{value}; \\
  \text{identifier} & : \text{type} = \\
  & \{ \\
  & \} \\
\end{align*}
\]

The type can be as simple as \textit{int} or more complex. Here are some basic examples of declarations:

\begin{verbatim}
my_integer : int;
my_return_forty_two : () -> int;
my_int_incr : (argument : int) -> int;
my_T_incr : [T : type](argument : T) -> T;
\end{verbatim}

Listing 2.1: Example of SCOOL declarations

There are two different kinds of function arguments: static and dynamic ones. Static arguments are equivalent to C++ templates whereas dynamic ones are classic function arguments. This will be further discussed in the chapter 4 on page 18, but what is important to notice here in a syntax-wise optic is that the hooks [] are dedicated to the static world and the parenthesis () are dedicated to the dynamic world.

The special type \textit{type} is used to declare a general template parameter (equivalent to \textit{class T} or \textit{typename T} in C++).
2.2 Verbatim C++

SCOOL does not aim at being a complete representation of C++. For example it is not interesting to implement `iostream` in SCOOL. The language provides the ability to include C++ directly. The code included is not parsed so it is literally copied to the destination source file, and therefore must be a valid C++ construction. The escape characters to include C++ are `|` and `|`. For example:

```cpp
|[ #include <iostream> ]|

f : (arg : int) -> int =
{
    |[ std::cout << arg << std::endl; ]|;
    -> arg;
}
```
Chapter 3

Types

3.1 Constness and references

In C++ the philosophy is that one needs to add explicit keywords to enable features. For example to have constant types, the \texttt{const} keyword is required. As this is coherent in the C-language world where everything is explicit, it is not from a SCOOL point of view.

In SCOOL all the types are read-only by default and additional type modifiers are required to get non-read-only types.

For example, the C++ code corresponding to the Listing 2.1 is:

```cpp
const int my_integer;
int my_return_forty_two();
int my_int_incr(int argument);

template <typename T>
my_T_incr(const T& argument);
```

As expected, the \texttt{my_integer} variable is constant, as the \texttt{argument} parameter of the \texttt{my_T_incr} function is. The parameter for the \texttt{my_int_incr} is not const because it is a primitive type and it is not useful to have a const-reference here.

The two qualifiers used for type modifications are \texttt{var} and \texttt{ref}. The \texttt{var} keyword is used to declare a mutable variables of a certain type, whereas the \texttt{ref} keyword is used to have a non-constant reference on a type. Here is an example of their usage:

```cpp
var my_integer : int;

my_int_incr : (argument: ref int) -> void;
```

The C++ translation is:
3.2 Type contraints: where clauses,

SCOOL provides the ability to constrain types thanks to *where* clauses. The syntax is quite straightforward:

\[ T : \text{type where} \langle \text{static\_condition} \rangle \]

The static conditions can be the same as the one of the static *if* statement (see subsection 4.3.2 on page 20).

The result is a constrained type that can be used wherever you can use a static parameter (basically in functions, see chapter 4 on page 18).

Note that this part of SCOOL is under very heavy work. Indeed, the dispatch algorithm is very complicated and requires a lot of meta-code. The section 8.5 on page 34 presents the actual state of *where* clauses in SCOOL.
Chapter 4

Functions

4.1 Generalities

There are mainly two kinds of functions. The first kind, dynamic functions, are the usual C++ functions: they can have arguments and template parameters and they return a value. The second kind, static functions, take only static parameters and return a type.

The general syntax of functions is:

```cpp
<qualifier> <identifier> : [ <static parameters> ] (<dynamic arguments>)
    -> <return type> =
    {
        // ...
        -> <return value>;
    }
```

The -> stands for the C++ return statement. There is a neat little sugar for one-line functions:

```cpp
<identifier> : [ <static parameters> ] (<dynamic arguments>)
    -> <return type>
    => <return value>;
```

4.2 Dynamic functions

The Listing 2.1 on page 14 shows examples of basic function declarations. Basically they are similar to classic C++ functions. Here are some more advanced function usages.

```cpp
// Function that takes no argument and returns an integer.
my_return_forty_two : () -> int =
    {
        42;
    }
```
// Function that takes an integer and returns its successor.
my_int_incr : (argument : int) -> int => argument + 1;

// Generalisation to any type of the function above.
my_T_incr : [T : type] (argument : T) -> T => argument + 1;

The C++ translation:

```cpp
int my_return_forty_two ()
{
    return 42;
}

int my_int_incr (int argument)
{
    return argument + 1;
}

template <typename T>
T my_T_incr (const T& argument)
{
    return argument + 1;
}
```

### 4.3 Static functions

For the sake of clarity and logic, this section is in the Function chapter. However, static functions use intensively classes and it is recommended to read and understand the class chapter before (chapter 5 on page 23).

The concept of static functions has already been brought up in the State of the Art chapter (see chapter 1 on page 6). The basic idea is that a static function is a function executed at compile-time taking only static parameters, which means either types or literal primitive values (int, float, ...), and returning a type.

Obviously, a such function is quite particular and cannot contain whatever a classic function can contain. Thanks to tools developed by the OLENA core team, SCOOL provides some static constructions (where clauses, if/else, ...). They can be combined to create very easily static functions that would require hundreds of classic C++ code otherwise.

#### 4.3.1 Declaration

There are two manners to return a type: either return a pre-existing one or create it on the fly. The two syntax are:
4.3 Static functions

4.3.2 Static if statement

The static if is a static statement which means it is resolved at compile time and has no runtime overhead. It is syntactically very close to the usual if, only using the hooks [] instead of the parenthesis ().

It can test some properties on types:

- Equality and difference
  
  \[
  \text{if} \ [ \ T = U \ ] \quad \text{if} \ [ \ T \neq U \ ]
  \]

- Inheritance relationship (does T inherit from U?)
  
  \[
  \text{if} \ [ \ T < U \ ]
  \]

- Concept modeling relationship (does T model U which must be a concept?)
  
  \[
  \text{if} \ [ \ T \text{ models } U \ ]
  \]

See the next sections for concrete examples of static if usages.

4.3.3 Returning pre-existing type

Here is an example of a function \( f \) always returning the \( \text{int} \) type:

\[
f : [ ] \rightarrow \text{type} = \\
| \rightarrow \text{int};
\]
Another example, the identity function:

```haskell
identity : [T : type] -> type =
{ -> T;
}
```

A more complicated example: a function taking two types and returning a type equal parameterized by the type if they are equal, and a type not_equal if they aren’t:

```haskell
are_equal : [T : type, U : type] -> type =
{ if [ T = U ] -> equal[T];
  else -> not_equal;
}
```

This could be very useful when writing meta-algorithms.

### 4.3.4 Creating a type on the fly

Now some examples of creating the return type on the fly. Obviously, it will always be a class.

First, a basic function taking one static parameter $T$ and creating a class containing a value of type $T$ and a type value_type equal to $T$:

```haskell
container : [T : type] -> class =
{ public
  { value_type : type = T; // Equivalent to C++ typedef

    var value : value_type;
  }
}
```

Listing 4.1: A simple container

The body of the function will literally be the body of the class. Though, you can have much more powerful static functions if you consider static statements, like the static if (see subsection 4.3.2 on page 20).

For example we could have a container that, for the \texttt{int} type is strictly the one listed in the Listing 4.1 and for all the other types have value being private and a getter and a setter defined:
container : [T : type] -> class =
{
    public
    {
        value_type : type = T;
    }
    if [ T = int ]
    {
        public
        {
            var value : value_type;
        }
    }
    else
    {
        public
        {
            get : () -> value_type => value_;
            mutable set : (value : value_type) => value_type = value;
        }
        private
        {
            var value_ : value_type;
        }
    }
}
Chapter 5

Classes

5.1 Generalities

As SCOOL is object-oriented it provides an implementation of the object model. Part of it works like the C++ model: classes, methods, access blocks… However, the implementation differs: the inheritance is static, types can be virtual…

The purpose of SCOOL’s object model is to bring all the power of the usual C++ one to the static world without any code overhead.

One part of the object model is classes. Their basic features almost work like C++ ones though there are little differences: methods are constant by default, there have no default accessibility…

5.2 Declaration

Declaring a SCOOL class is quite straightforward:

```plaintext
foo : class =
{
}
```

5.3 Access blocks

There is no default accessibility in SCOOL so it is mandatory to explicitly define the current access block to be either public, protected or private. They strictly have the same meaning as the C++ ones though their syntax differ to be more coherent with the concept of blocks. Like in C++ there is no problem having multiple times the same access block, but it is strictly forbidden to have nested blocks.
The block style is inherited from C++ with a more coherent syntax though. It might not be relevant as the accessibility property is rather linked to an entity rather than a block. It could be very easy to add the JAVA-style syntax to the language.

## 5.4 Methods and variables

Access blocks can contain type definitions, variables or methods. To respect the SCOOL’s philosophy of read-only by default, methods are constant by default and must be explicitly qualified as mutable if they are not.

```plaintext
point1d : class =
{
  public
  {
    point_type: type = int;
    get_x : () => point_type => x;
    mutable set_x : (x : point_type) => point_type => x_ = x;
  }
  private
  {
    var x : point_type;
  }
}
```

## 5.5 Constructor and destructor

The constructor and the destructor are respectively the special methods make and destroy. They behave like in C++. They have a noticeable difference with other SCOOL methods: they do
not return a value and so, do not have a return type (not even `void`).

```plaintext
point1d : class =
{
    public
    {
        point_type : type = int;
        make : () = { x_ := 0; }
        make : (x : point_type) = { x_ := x; }
        destroy : () = { x := -1; }
    }
    private
    {
        var x_ : point_type;
    }
}
main : () -> int =
{
    p : point1d = make(42);
    -> 0;
}
```

### 5.6 Inheritance

The SCOOL's class inheritance is not the usual C++ one. Indeed, it is static. This means that there is strictly no runtime overhead due to dynamic dispatch. However, if we do not use the C++ dynamic dispatch, we still want a similar feature in a static flavor. We therefore have to provide some little meta-code that performs the static dispatch. All this process is completely transparent in SCOOL and there is no code overhead to benefit from the static inheritance.

Here is a basic example:

```plaintext
animal : class =
{
}

cat : class < animal =
{
}
```

It is then possible to use a such hierarchy:
# 5.7 Parameters

Refer to the section *Static functions* (section 4.3 on page 19) for details on parameterized classes. In SCOOL the usual parameterized C++ classes are seen as static functions returning a class created on the fly (see subsection 4.3.4 on page 21).

```cpp
// [include <iostream>]

Animal : class =
{
  decl scream : () -> void;
}

Cat : class < Animal =
{
  scream : () -> void = [ std::cout << "Miaou" << std::endl; ];
}

Dog : class < Animal =
{
  scream : () -> void = [ std::cout << "Wouf" << std::endl; ];
}

scream : (animal : Animal) -> void =
{
  animal.scream();
}

main : () -> int =
{
  var cat : Cat;
  var dog : Dog;

  scream(cat); // Prints 'Miaou'
  scream(dog); // Prints 'Wouf'

  -> 0;
}
```

Listing 5.1: Animal hierarchy

Refer to the Listing 7.1 on page 30 to have the C++ translation.
Chapter 6

Concepts

6.1 Generalities

A weakness of C++ is that it is not possible to express constraints when using generic programming. Typically constraining the types of template parameters is impossible which leads to insecure code and some very obscure error messages when wrong types are used. A solution to this problem is concepts. They allow one to add constraints to template parameters so that they must model some concepts. This approach guarantees a much more secure code and clear error messages.

6.2 Basics

SCOOOL provides such concepts (Géraud and Levillain, 2008) with the ability to constrain static parameters. They can ensure that type definitions or methods in a given type exist. For example, consider the Box concept that checks if there is a value_type type definition and two methods: get_value and set_value.

```
Box : concept =
{
  value_type : type;

  get_value : () -> value_type;
  mutable set_value : (value : value_type) -> void;
}
```

Listing 6.1: Example of concept declaration

It is then possible to use this concept to constrain static parameters:

```
f : [T : type where (T models Box)] (val : T) -> void =
{
  point : T::value_type = val.get_value();
}
```

Listing 6.2: Example of concept usage
6.3 C++ translation

SCOOL concepts are not independent as C++0x ones. In fact, they strongly rely on what they actually are: classes. This implies that it is mandatory for a type to declare what concept it models because of the underlying class inheritance relationship. For example:

```cpp
Element : class models Box =
{
    public
    {
        value_type : type = int;
        get_value : () => value_type => value_;
        mutable set_value : (value : value_type) => void =
        {
            value_ := value;
        }
    }
    private
    {
        value_ : value_type;
    }
}
```

Another consequence of the way concepts are implemented is that it is impossible for a class to model multiple concepts.

### 6.3 C++ translation

As SCOOL targets C++ 2003 it does not take advantage of the concepts found in the next C++ norm. We had to find a trick allowing us to have the ability to statically check our concepts without runtime overhead. The trick was found by the OLENA core team (Géraud, 2006) and consists in having a parameterized class by concept and performing the checks in its constructor. The parameter of the concept class will be the type to check. To ensure the presence of a type, we will try to typedef it, and for a method we will try to get a pointer on it. All these checks are performed at compile time and do not impact at all on the runtime.

The C++ translation for the Listing 6.1 is:

```cpp
template <typename E>
class Box
{
    // To be provided by classes that model this concept.
    // typedef value_type;
    // const value_type& get_value();
    // void set_value(const value_type&);
    protected:
        Box();
};
```
The C++ translation for the Listing 6.2 is:

```cpp
template< typename E >
Box<E>::Box()
{
    typedef typename E::value_type value_type;

    value_type (E::*m1)() const = &E::get_value;
    m1 = 0;

    void (E::*m2)(const value_type&) = &E::set_value;
    m2 = 0;
}
```

The exact routine, developed by the OLENA core team, finds the exact type from an object. In this case, it will return the type $T$. 

Chapter 7

Examples and translations

7.1 Introduction

In this section you will find some examples that enlighten the power of SCOOL and their translation into C++.

7.2 Simple hierarchy

The translation of the listing Listing 5.1 on page 26 is:

```cpp
#include <iostream>

template <typename T>
class Animal
{
  void scream()
  {
    const T& self = static_cast<const T*>(this);

    self.scream();
  }
};

struct Cat : public Animal<Cat>
{
  void scream() const
  {
    std::cout << "Miaou" << std::endl;
  }
};
```
```cpp
struct Dog : public Animal<Dog>
{
    void scream() const
    {
        std::cout << "Wouf" << std::endl;
    }
};

template <typename E>
void scream(const Animal<E>& a)
{
    a.scream();
}

int main()
{
    Cat c;
    Dog d;

    scream(c);
    scream(d);

    return 0;
}
```

Listing 7.1: Animal hierarchy
Chapter 8

Future work

8.1 Classes

8.1.1 Object construction

Often, to be correctly constructed, objects require the construction of the other objects it contains with custom constructors. Typically:

```cpp
class A
{
  public:
    A() : b(/* some stuff... */) {
      // ...
    }
  private:
    B b;
};
```

There is currently no way to represent this construction in SCOOL and it will require a syntax extension. It would likely look like:

```cpp
A : class =
{
  public
  {
    make : () =
    {
      initialize
      {
        b := make(/* some stuff... */);
      }
      // ...
    }
  }
}
```
8.2 Concepts

8.2.1 Multiple concept modeling
For now it is impossible for a type to model multiple concepts, though this could be an interesting feature which would require very few syntax extensions.

This should be further discussed with the OLENA core team.

8.2.2 Static SCOOL check algorithm
To rely less on the concept implementation in C++, we could provide extended concepts that are not translated into C++ but that are statically checked by the SCOOL translator. Obviously, this is not that simple to implement and would probably require type checking.

8.3 Type checking
As SCOOL has more and more features it would be very interesting to have type checking inside the language. This would allow very powerful features such as:

- Static check of the program
- More powerful concepts
- More complex code constructions
- ...  

However, the type checking algorithm is clearly not obvious and will require a lot of work especially to handle where clauses (see section 3.2 on page 17).

8.4 Static statements
SCOOL handles basically one static statement: the if. Though, more statements could be added, like type pattern matching. This would be very powerful but also very complicated because it would rely, as the static dispatcher of the where clauses (see section 8.5 on page 34). The syntax could look like:
8.5 Where clauses - static dispatcher

Where clauses are one of the interesting features of SCOOL but also one of the more complicated. Indeed, they are used in the static parameters of functions (see chapter 4 on page 18) and we want to have a static dispatch based on them (as there is a dynamic dispatch when overloading classical dynamic functions).

SCOOL must provide the static dispatch meta-algorithm but it is obviously not easy to write. A first version which should be used as starting point has been written by Thierry Géraud (see Listing A on page 37).

8.6 Functions

As one of the key features of SCOOL is static function overloading it might be interesting to have other methods to specialize parameters than the usual position specialization. Typically the name specialisation could be interesting, though this requires some advanced treatments from the SCOOL translator.

8.7 Grammar and translator improvements

The grammar was written by Quentin Hocquet for his first work on SCOOL and did not evolve with the objectives of SCOOL. The current situation is a mix between the original grammar and extended parts to implement the new features and leads to incoherences. To achieve a stable state, it is mandatory to rewrite the grammar nearly from scratch.
Conclusion

Through this report we saw a lot of interesting SCOOL features. Compared to the other languages presented in the State of the Art (chapter 1 on page 6), SCOOL has clearly some interests. Its conception of static functions (section 4.3 on page 19), of where clauses with static dispatch (section 3.2 on page 17), of concepts (chapter 6 on page 27), …, have all their own strengths and drawbacks. The main advantage residing in the fact that all those features are brought to C++ and have no runtime overhead.

As SCOOL evolves, it becomes closer and closer to a stable state were we will be able to make a first release. We will then be able to start using it more widely for papers and for projects.
Bibliography


Appendix A

Static dispatcher

```cpp
struct foo_1 {};  // foo : [ T : type ] -> class =  
struct foo_2 {};  //  {  
struct foo_3 {};  //   -> select  
    foo_1 when (T = float)  
    foo_2 when (T = int)  
    foo_3 when (true)  
};

template<typename T1, typename T2>
struct cmp
{
    enum { eval = false };  
};

template<typename T>
struct cmp<T, T>
{
    enum { eval = true };  
};

template<unsigned i, typename T>
struct test_foo;

template<unsigned i, typename T>
struct result_foo;

template<typename T>
struct test_foo<1, T>  
```
```cpp
[ 
  enum { eval = cmp<T, float>::eval }; 
];

template <typename T>
struct result_foo<1, T>
{
  typedef foo_1 ret;
};

template <typename T>
struct test_foo<2, T>
{
  enum { eval = cmp<T, int>::eval }; 
};

template <typename T>
struct result_foo<2, T>
{
  typedef foo_2 ret;
};

template <typename T>
struct test_foo<3, T>
{
  enum { eval = true }; 
};

template <typename T>
struct result_foo<3, T>
{
  typedef foo_3 ret;
};

struct not_found;

template <typename T>
struct found
{
  typedef T unwrap;
};
```
template <unsigned i, bool b, typename T>
struct run_eval_foo;

template <unsigned i, typename T>
struct run_eval_foo<i, false, T>
{
  typedef not_found ret;
};

template <unsigned i, typename T>
struct run_eval_foo<i, true, T>
{
  typedef found<typename result_foo<i,T>::ret> ret;
};

template <typename R1, typename R2>
struct merge_eval;

template <typename R1, typename R2>
struct merge_eval<found<R1>, found<R2> >
{
  typedef found<R1> ret;
  // bug: two matches (at least)
};

template <typename R>
struct merge_eval<found<R>, not_found >
{
  typedef found<R> ret;
};

template <typename R>
struct merge_eval<not_found, found<R> >
{
  typedef found<R> ret;
};

template <>
struct merge_eval<not_found, not_found >
{
  // bug: nothing found
};
template <unsigned i, typename T>
struct eval_foo {
  typedef typename
    merge_eval<
      typename run_eval_foo<i, test_foo<i,T>::eval, T>::ret,
      typename eval_foo<i+1, T>::ret>::ret ret;
};

template <typename T>
struct eval_foo<4, T> {
  typedef not_found ret;
};

template <typename T>
struct foo {
  typedef typename eval_foo<1, T>::ret eval;
  typedef typename eval::unwrap ret;
};

#define foo(T) foo<T>::ret

int main() {
  std::cout << typeid(foo(int)).name() << std::endl;
}