

Introducing Scool

Maxime van Noppen¹, Roland Levillain^{1,2}, Akim Demaille¹

¹ EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire; FR-94276 Le Kremlin Bicêtre; France
`maxime.van-noppen@lrde.epita.fr`, `roland.levillain@lrde.epita.fr`,
`akim.demaille@lrde.epita.fr`

² Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Équipe A2SI,
ESIEE Paris, Cité Descartes, BP 99, FR-93162 Noisy-le-Grand Cedex, France

Abstract. C++ has proved to be a powerful language to write generic and efficient libraries. However using classical Object-Oriented (OO) C++ does not suffice to reach the levels of efficiency required in domains where large data sets have to be processed through generic algorithms, e.g., scientific libraries. A solution consists in combining the power of Object-Oriented Programming and *static* programming—which is in fact meta-code expressed thanks to C++ template constructs. This has the advantage of replacing the OO run-time overhead (due to virtual method dispatch) by compile-time computations. However, such an approach relies on code that is verbose, hard to write and to maintain. Though powerful, C++ lacks high-level static features, and thus clutters the semantics of static constructs with unrelated code.

We present SCOOOL, a “static” language mixing OO and Generic Programming (GP) created to take advantage of all the power of static C++ thanks to a more expressive syntax and high-level constructs, without the drawbacks of plain C++. As a full-fledged static OOP language, SCOOOL provides polymorphic methods (i.e., inclusion polymorphism), with the notable difference that every polymorphic call is statically-resolved: the design of SCOOOL is based on the property that the exact (dynamic) type of every object is known at compile-time.

As the aim of SCOOOL is to bring all the power of static Object-Oriented Programming to C++, it is not directly compiled but translated into human-readable C++. The development of the translator raised classical problems found in Domain Specific Languages (DSLs) like traversal strategies of the abstract syntax tree. We propose an original solution based on the Stratego/XT program transformation framework, in the context of MILENA, a generic and efficient C++ library from the OLENA image processing platform.

1 Introduction

When designing software libraries, especially in the field of scientific computation, one aims at two major objectives simultaneously: *efficiency* because of the large data sets to be processed, and *genericity* since algorithms can often be applied to many types of data. To fulfill these requirements, the C++ language

[1] has proved to be an appropriate tool [2,3,4,5]. As the language offers several programming paradigms (procedural programming, Object-Oriented Programming (OOP), GP, and some kind of Functional Programming (FP)), it allows programmers to express their programs using many *abstractions* (functions, abstract classes, concepts, functors, libraries, etc.). Abstractions are fundamental in programming to carry out desirable qualities in software engineering [6,7].

Reusability Well-designed abstractions are reusable in essence: they allow the programmer to reuse existing algorithms or data structures with new code, provided existing interfaces are respected.

Factoring Reifying common parts into abstractions reduces the amount of code, hence the number of potential bugs. Factoring occurs at least on two levels. On the one hand existing abstractions can provide general or generic code, used in new data structures based on old ones. On the other hand, programming with abstractions encourages programmers to design new code along existing high-level constructs, so that their additions can benefit to the next users of the same code base.

Flexibility Designing orthogonal abstractions helps to reduce code overlapping and coupling between two entities (modules, libraries, classes, etc.). This favors code reuse, lowers dependencies among a project and helps to maintain separate compilation.

Closeness to Theory Code written with abstractions is free of implementation details and closer to notations used in scientific fields (mathematics, logic, image processing, etc.).

However, abstractions often come at the cost of efficiency. A canonical example is the use of polymorphic methods (C++'s virtual member functions). In most languages—in particular in C++—a polymorphic call is more expensive at run time than a non-polymorphic one. Moreover, the late binding of such a method prevents optimizations such as inlining, constant propagation, etc. Another example is code matching mathematical notations with disastrous performances (both in space and in time) because of numerous temporary objects [8]. In domains such as scientific computations where large data sets are to be processed by generic algorithms, such a loss is not acceptable.

To solve this issue, several programming techniques have been developed, based on the use of metaprogramming, consisting in complex template code implementing usual OOP-like features, but in a *static* flavor. The drawback of this approach is that dynamic features that used to be simple become tricky to implement, along with support code being larger and harder to read. Libraries have been created [9,10] to work with static idioms and factor reusable code.

Despite their usefulness, these libraries cannot fully make up for the lack of powerful high-level static constructs built in C++. Integrating these features into the language itself would be convenient and useful to spread the use of static programming techniques. In the case of the MILENA generic and efficient C++ library from the OLENA image processing platform [11], we studied different ways to solve this issue. The first approach was to integrate a DSL inside C++.

The solution relied on the creation of a generic source-to-source C++ transformation platform, TRANSFORMERS [12]. Among the goals of TRANSFORMERS is the ability to equip C++ with new keywords and constructs such as syntactic sugar for existing, hard to write standard C++ code. A first requirement of this approach is to fully handle both parsing and type checking of standard C++. It turned out to be harder than expected [13], therefore we designed another solution: extend C++ “from the outside”, wrapping it into a new DSL. The Static Object-Oriented Language (SCOOL) was created to address issues regarding the *static* world by providing easy-to-use high-level *static* constructs [14,15]. The goal of SCOOL is to provide a simple way to prototype static C++ programs rather than writing actual C++ code. Therefore SCOOL programs are translated into C++ which may use some of the aforementioned libraries. Developed using STRATEGO/XT [16], the translator benefits from a high degree of modularity. This leaves the door open to multiple back ends and targets in the future.

This paper is structured as follows: Section 2 explains the goals of SCOOL and addresses the problem of creating a DSL extending C++. Prominent aspects and features of SCOOL are detailed in Section 3 while the implementation of the SCOOL front end is sketched in Section 4. Section 5 presents related work and Section 6 concludes.

2 C++ for Intensive computations

2.1 Scientific Libraries

Scientific libraries are particular regarding to their algorithms and data types. Indeed, types are mapped from mathematical abstractions that are to be handled by generic algorithms. Classical object-oriented programming fulfills the design needs by making possible the expression of complex hierarchies via classes, inheritance (possibly multiple), overloading... However, these hierarchies incur a run-time overhead being the dynamic dispatch. If this is generally an acceptable cost, in this particular field it is far too expensive.

Another tool for generic programming is *templates*. As in C++ they are static (resolved at compile-time) they have no run-time cost yet they make possible the writing of completely generic algorithms. This very high degree of genericity comes at the price of a wordy and intricate syntax, as well as complex and lengthy error messages from the compiler.

Schematically, providing S structure types, V value types, and A generic algorithms—i.e $S + V + A$ entities—a library features $D = S \times V$ different data sets (input types) and $D \times A$ possible processing routines. To avoid the combinatorial explosion, using generic algorithms (possibly specialized if needed) is mandatory. However, classical GP is of no use here as the algorithms have to be strongly typed.

These issues emerged essentially during the development of two scientific libraries: VAUCANSON (finite state machines [17]) and OLENA (image processing [18]). The solution found for these libraries was to merge the two worlds: OOP

and static programming via a new paradigm: SCOOP [19]. It was then refined and enhanced in a second version [10]. The core idea was to emulate OOP in the static world thanks to the Curiously Recurring Template Pattern [20] and a static dispatch. This is possible thanks to the underlying property that the exact type of every object is known at compile-time. Furthermore we still have the ability to write generic and efficient algorithms, strongly typed, in an OOP flavor.

2.2 C++ as a Reference Language for Scientific Computing

The question of the implementation language is an important matter when designing software. In the context of scientific computing, aforementioned qualities (programming with abstractions, static constructs and efficiency) are vital. Though other languages (see Section 5) offer interesting static features, C++ is currently the best trade-off regarding multiple criteria:

- it is a standard [1];
- it is efficient (regarding run-time performance) for scientific libraries;
- it allows complex type hierarchies;
- it allows the writing of efficient generic algorithms;
- it is a widespread and well known language, benefiting from a large community.

Though C++ is not the best choice for each single criterion, it is in our experience, the best overall compromise. This requires intensive use of `templates`: the code may be very large and complicated, leading to complications for maintenance and readability. Furthermore, as C++ lacks high-level static abstractions, the code is often not correlated with its semantics. Consider the following classical example:

```

template <unsigned N>
struct fact
{
    enum { res = N * fact<N - 1>::res };
};

template<>
struct fact<0>
{
    enum { res = 0 };
}

```

This code is representative of the gap between the implementation and the semantics. The meta-code is a simple recursive function that computes a factorial at compile-time. A dynamic equivalent is:

```

unsigned fact(unsigned N)
{
    if (N == 0)
        return 1;
    return N * fact(N - 1);
}

```

The syntax of both versions is very different although they have similar semantics. In the latter simple idioms appear clearly: there is a function taking one integer argument, checking its value and returning the computation result. This is completely obfuscated in the former example.

Some languages targeting C++ [21] (see also Section 5.2) feature a mapping between high-level idioms (functions, `if`, loops, etc.) and their static counterparts in C++. The following table maps classical dynamic statements to static equivalent constructs and shows how dynamic features gets corrupted when converted to static ones.

High-level idiom	C++ static counterpart
Function	Parameterized class
<code>if</code>	Class specialization
Loop	Recursively-defined class template
<code>return</code>	Public <code>typedef</code> of a class template

The goal of SCOOL is to remedy to this issue by providing an easy way to write static code. The equivalent SCOOL code for this example is:

```
fact : [N : unsigned] → unsigned =
{
  if [ N == 0 ]
    → 1;
  → N * fact[N - 1];
}
```

2.3 Extending C++

There are several ways to enhance C++ with advanced static features. The first step is to build a library, or meta-library, since it offers services to build statically-enhanced libraries [19,10]. However, libraries does not provide easy-to-use and concise high-level static constructs to the user. These features should also be available from the language layer.

Libraries Extending C++ using an ad language is a costly decision. Designing, developing, documenting, testing and maintaining a new language with the required tools is tempting but doomed to be failure in mid-term in many cases [22]. Instead, building the needed extensions as a library on top of a general-purpose language, and adding some syntactic sugar to this language, is a cheaper and sounder strategy.

In the case of the OLENA image processing platform, built following the principles of the Static C++ Object Oriented Programming (SCOOP) paradigm [19,10], two libraries were developed to express and factor static constructs: STATIC and METALIC. OLENA is based on top of these libraries since it replaces several mechanism built in C++ (explicit (named) inheritance, dynamic dispatch of methods, classic inclusion polymorphism) by its own (implicit (property-based) inheritance, static dispatch of methods, statically-constrained inclusion polymorphism).

Extending C++ from the inside In order to simplify OLENA’s implementation, the design of a DSL has been evoked (and sometimes tried) many times. First attempts have dealt with unsuccessful C++ extensions. The first idea was to develop a general platform for C++ transformation [23,24,12,25,13], namely the TRANSFORMERS project. As time passed, it came to light that this task was much more difficult than expected. Other existing C++ program transformation platforms were considered (e.g. CodeBoost [26]), but as they did not strictly follow the grammar of the ISO/IEC C++ 2003 standard (a requirement of the OLENA Team regarding the evolution of the project), those were left out.

Extending C++ from the outside Consequently, an opposite design was chosen. As it is indeed easier to include C++ in a specific DSL, the idea of a language able to embed C++ bits (and still targeting C++) was examined. It appeared that *sugaring* C++ from the outside using a brand new syntax offered many advantages:

A better syntax Developing a new DSL enables new syntactic constructs, more consistent, and more expressive regarding static features in our case.

A much faster development time Not having to deal with C++ parsing and type-checking issues is a real relief for the authors of this DSL. As a new language, it can benefit from sound syntactic and semantic specifications.

Freedom with respect to the existing standard As C++ is embedded into the new language, parsing problems are postponed to the C++ compiling stage. The responsibility of the semantic analysis of C++ parts is thus left to the C++ compiler. This is acceptable since the roles are well separated between the DSL and C++: the former handles the “skeleton” aspects (structure), while the latter is responsible for the “flesh” of algorithms (functions’ and methods’ bodies).

Better extensibility In spite of C++’s dominant position in scientific computing (along with other languages like C and Fortran), OLENA might consider another language someday, better suited to image processing problems, if one was to replace C++. Though not automatic, having a DSL loosely coupled to C++ allows other targets as well. Even with C++ as a target, this DSL would make the transition of OLENA’s SCOOOL code base to the next standard [27] easier.

The compiler from this DSL to C++ must be much more than a preprocessor. First of all, it should come with its own semantic analyzer, so as to avoid extra errors on the generated code and to produce relevant error messages. As this compiler does not feature a full C++ front-end (this lack of requirement being a motivation for this solution), it cannot report errors on embedded C++ directly. However, as SCOOOL handles most of the complex structures, including most **templates**—i.e. the ones that are likely to produce complex C++ error messages—this limitation is not unacceptable. Secondly, it should generate high-quality (pretty-printed) C++, as these outputs are not only to be compiled, but

```

// An integer.
foo : int = 42;

// A function with no arguments and returning an int.
bar : () → int;

// A function with an int argument returning and int.
baz : (arg : int) → int;

// A function parameterized with a type T, with an argument of type T and returning a T.
qux : [T : type](arg : T) → T;

```

Listing 1. An example of some SCOOL declarations.

also read and used by humans as well. Last, the compiler should be easy to extend, since prototyping is one of the goals of such a DSL.

We present below the SCOOL language, a DSL aimed at writing static C++ code easier, in particular within the OLENA project.

3 The SCOOL programming language

3.1 Syntax basis

Being a brand new language, SCOOL does not make the same syntactic and semantic sacrifices as C++. All declarations (classes, functions, variables, concepts, etc.) follow the same syntax, which is:

```

⟨ identifier ⟩ : ⟨ type ⟩ ;
⟨ identifier ⟩ : ⟨ type ⟩ = ⟨ value ⟩ ;
⟨ identifier ⟩ : ⟨ type ⟩ = { /* ... */ }

```

The *type* denote basic (e.g. `int`) or complex types (functions, classes, concepts...). Listing 1 shows some basic examples of declarations. All the information regarding the type of the declared objects is at the same place, denoted with a functional programming style. This differs from C++ where for instance function types are split in at least two parts, the return type and the arguments being separated by the function identifier. The static and dynamic worlds have their own and distinct syntax. Brackets `[]` are exclusively used for static code whereas parenthesis `()` are dedicated to the dynamic code. This is well illustrated by the `if` statement which provides both flavors:

```

if (foo = 42) // Run-time conditional.

if [T = int] // Compile-time conditional.

```

Finally, the special type *type* is used to declare a *type variable* which is basically a read-only variable denoting a type. Depending on the context it can be equivalent to a C++ `typedef` or `typename` parameter.

3.2 Embedding C++

SCOOOL does not aim at being a complete representation of C++. For example it is not interesting to implement `std::ostream` in SCOOOL. Thus it is possible to include verbatim C++ code. Such code is literally copied to the destination source file, and therefore must be valid C++. The escape characters used to embed C++ are `[` and `]`. Verbatim C++ blocks may comprise whole statements or just an expression. For instance:

```

[[ #include <iostream> ]]

f : (arg : int) → int =
{
  // Escape an entire statement.
  [[ std::cout << arg << std::endl; ]];

  // Escape only an expression used in a Scool construct.
  // '→' is equivalent to C++'s 'return' keyword
  → [[ arg + 42 ]];
}

```

3.3 Types

Constness and references In SCOOOL all types are read-only by default and type modifiers are required to get read-write access to values. This enforces safety safer code as extra-constness does not harm. For example, the C++ code corresponding to Listing 1 is:

```

const int foo = 42;

int bar();
int baz(int arg);

template <typename T>
T qux(const T& arg);

```

The `bar` variable is constant, as is the `arg` formal argument of the `qux` function. The formal argument of `baz` is not constant because it is a primitive type and it is not useful to have constant references here.

The `var` keyword declares mutability (`var foo : int;` is translated as `int foo;`), and `ref` constructs non-constant references (`increment : (arg: ref int) → void;` is translated as `void increment(int& arg);`).

Type constraints: where clauses SCOOOL provides the ability to constrain types thanks to `where` clauses:

```

T : type where ( static_condition )

```

Static conditions are introduced later, during the presentation of the static `if` statement (see paragraph 3.6). The result is a constrained type that can be used wherever you can use a static parameter (basically in functions, Section 3.4). This part of SCOOOL is under very heavy work. Indeed, the dispatch algorithm is very complex and requires a lot of meta-code.

3.4 Functions

There are two kinds of functions. *Dynamic* functions are akin to C++ functions: they have arguments (and possibly template parameters) and return a value. *Static* functions take only template parameters and return a type. The general syntax is:

```

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

```

Listing 1 shows examples of basic function declarations. They are similar to classic C++ functions. The following listing shows some more uses of functions, with and without static parameters.

```

// A function taking no argument and returning an integer .
my_return_forty_two : () → int =
{
  → 42;
}

// A function taking an integer and returning its successor .
my_int_succ : (arg : int) → int =
{
  → arg + 1;
}

// Generalization to any type of the function above.
my_T_succ : [T : type] (arg : T) → T =
{
  arg + 1;
}

```

The corresponding C++ translation is as follows.

```

int my_return_forty_two()
{
  return 42;
}

int my_int_succ(int arg)
{
  return arg + 1;
}

template <typename T>
T my_T_succ(const T& arg)
{
  return arg + 1;
}

```

3.5 Classes

SCOOL's object model is similar to C++'s one: classes, methods, access blocks, etc. However, the implementation differs: inheritance is static, types can be `virtual`, etc. Its purpose is to bring all the power of the usual C++ model to the static world without any code overhead.

One part of the object model is classes. Their basic features almost work like C++ ones except for some aspects: methods are constant by default and have no default accessibility.

Declaration Declaring a class is straightforward:

```
foo : class = { /* ... */ }
```

Access blocks There is no default accessibility in SCOOOL: it is mandatory to define the current access block to be either **public**, **protected** or **private**. These blocks have the same semantics as in C++, yet their syntax differ to be more consistent with the concept of blocks. There can be any number of access blocks; their nesting is forbidden. A Java-like syntax is also being considered.

```
foo : class =
{
  public { }
  private { }
  public { }
}
```

Methods and variables Access blocks can contain definitions of types, variables or methods. To respect SCOOOL's "read-only by default" philosophy, methods are constant except if they are qualified as **mutable**.

```
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; }
}
```

Constructors and destructor The constructors and the destructor are respectively the special methods **make** and **destroy**. They behave like their C++ counterpart. They have a noticeable difference with other SCOOOL methods: they do not return a value and so, do not have a return type (not even **void**).

```
point1d : class =
{
  public
  {
    point_type: type = int;
    make : () = { x_ := 0; }
    make : (x : point_type) = { x_ := x; }
  }
  private { var x_ : point_type; }
}
```

```

main : () → int =
{
  p : point1d = make(42);
  → 0;
}

```

Inheritance SCOOL's class inheritance is not the usual C++ one. It actually is *static*. This means that there is strictly no run-time overhead due to dynamic dispatch. However, even 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 meta-code performing the static dispatch. All this process is completely transparent in SCOOL and there is no code overhead, not to spoil the benefits of static inheritance. In the following excerpt, “<” denotes an inheritance relationship.

```

[
  #include <iostream>
  #include <string>
  using std::string;
]

Animal : class = { public { decl talk : () → string; } }

Cat : class < Animal = { public { talk : () → string = { → "Meow!"; } } }
Dog : class < Animal = { public { talk : () → string = { → "Woof!"; } } }

talk : (animal : Animal) → void = {
  [ std::cout << animal.talk() << std::endl; ]
}

main : () → int =
{
  var cat : Cat; talk(cat); // Prints 'Meow!'.
  var dog : Dog; talk(dog); // Prints 'Woof!'.
  → 0;
}

```

Parameterized classes: static functions A static function is a function executed at compile-time taking only static parameters, which means either types, literal primitive values or expressions assessable at compile-time, and returning a type. Obviously, such a function is quite particular and cannot contain everything a classic function may contain. Thanks to tools developed by the OLENA Team, SCOOL provides some static constructs (**where** clauses, **if/else**, ...). They can be combined to easily create static functions that otherwise would require hundreds of lines of classic C++.

There are two ways to return a type: either return an existing one or create one on the fly. The example below presents both ways.

```

// Returning a preexisting type.
⟨ identifier ⟩ : [⟨ static.arguments ⟩] → type =
{
  // Static body.
  → ⟨ return.value ⟩ ; // Could be 'int', or any other type.
}

```

```

// Creating the type on the fly.
< identifier > : [ < static.arguments > ] → class =
{
  // Class body and/or static body.
}

```

Creating a type on the fly Listing 2 shows an example of definition of a function taking one static parameter T and creating a class containing a `value` of type T and a type `value_type` equal to T.

```

container : [T : type] → class =
{
  public
  {
    value_type : type = T; // C++ counterpart: typedef T value_type;
    var value : value_type;
  }
}

```

Listing 2. 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 static `if` (see Section 3.6). For example we could have a container that, for the `int` type is strictly the one listed in the Listing 2 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_ := value; };
    }
    private
    {
      var value_ : value_type;
    }
  }
}

```

3.6 Static if

A static `if` is resolved at compile time, it has no run-time overhead. It is syntactically close to the usual `if`, but uses square brackets instead of parentheses. It can test some properties on types:

```
// Equality and difference .
if [ T = U ] // ...
if [ T != U ] // ...
// Inheritance relationship (does T inherit from U?).
if [ T < U ] // ...
// Concept modeling relationship (does T model the concept U?).
if [ T models U ]
```

Figure 3 shows another example using static `if`.

```
// Always return the "int" type.
f : [] → type = { → int; }

// The identity on types.
identity : [T : type] → type = { → T; }

// Return the types "equal[T]" or "not_equal" depending on whether T and U are equal types.
are_equal : [T : type, U : type] → type =
{
  if [ T = U ]
  → equal[T];
  else
  → not_equal;
}
```

Fig. 3. Functions returning an existing type.

3.7 Concepts

C++03 provides no simple means to express constraints when using GP. Enforcing types verifications on a template by hand still depends on implicitly type-checked parameters and sometimes obscure error messages when the template's inputs are wrong. An approach is to express constraints on (static) parameters as *concepts*, i.e. a set of syntactic requirements over one or several types. However, concepts does not exist as actual entities in C++03. Reified concepts from C++0x offer a solution to this problem: template parameters can be forced to respect (or *model*) some concepts. This guarantees safer code and clearer error messages. SCOOL tries to mimic such a feature by providing syntactic constructs for concepts, and generating concept-checking code compatible with C++03. When the tools are mature enough to provide strong verification of types on SCOOL code, these concepts will also be checked *before* by the SCOOL translator.

Overview SCOOL provides concepts [10] with the ability to constrain static parameters. They can ensure that type definitions or methods exist within a

given type. For instance, the `Box` concept below checks that 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 4. An 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 5. An example of concept usage.

SCOOOL concepts are not specific entities as C++0x ones will be. In fact, they strongly rely on what they actually are as generated code: concept-checking classes (see below). This implies that a type must declare which concept it models because of the underlying class inheritance relationship. For example:

```
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 this implementation of concepts is that a class cannot model multiple concepts (yet).

C++ translation As SCOOOL targets C++03 it does not yet take advantage of reified concepts from C++0x. We had to find a means to statically check our concepts without run-time overhead. It consists in having one parameterized class per concept, enforcing checks in its constructor [28,29]. The parameter of the concept class is the type to check. To ensure the presence of a type, the code tries to create a `typedef` from it, and for a method it tries to form a pointer from it. All of these verifications are performed at compile time and have no impact on run time. The C++ translation for Listing 4 is as follows.

```
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();
```

```

};

template <typename E>
Box<E>::Box()
{
  // Concept checks.
  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 C++ translation for Listing 5 is:

```

template <typename T>
void f(const Box<T>& val_)
{
  const T& val = scool::exact(val_);

  typename T::value_type point = val.get_value();
}

```

The `scool::exact` routine, which is part of the STATIC “meta-library” developed for the OLENA Project, finds the exact type of an object. In this case, it would return the type `T`.

4 Implementation

The choice of STRATEGO/XT as a development platform for SCOOL was quite natural as it was already used for the TRANSFORMERS project. Moreover it actually fitted our needs regarding SCOOL’s development. We wanted to focus on three main points, namely:

- SCOOL’s grammar;
- high-level translations from SCOOL to C++;
- having a nice C++ pretty-printer to generate human-readable source files.

STRATEGO/XT allows us not only to focus on these points but also provides a very modular platform. SCOOL is in fact a collection of tools piped one into another, rather than a big monolithic bloc. The first step is the parsing which is performed thanks to a Scanner-less Generalized LR [30] parser using a Syntax Definition Formalism [31] grammar. Then the Abstract Syntax Tree (AST) is passed to the translator where the computations take place. It outputs a pseudo-C++ AST that is finally pretty-printed into C++. These components are completely separate modules that can easily be replaced. For example, one may want a D translator and pretty-printer, while still using the original parser module.

The translator is the most important part of SCOOL. It is entirely written in STRATEGO [32], a DSL for program transformation. A STRATEGO program consists of a set of rewrite strategies that are applied to an AST. If this approach perfectly handles simple node-to-node transformations, this is not enough for more complicated ones. This classical problem can be solved by executing two

passes on the AST: the first one to tag nodes (especially those requiring extra-work), and the second one to apply actual transformations. The solution that we retained is based on STRATEGO’s ability to dynamically define rewrite strategies [33]. Therefore the usual two passes are replaced by a single pass where all simple nodes are transformed while the others only generate strategies which will trigger real transformations when applied later on.

5 Related Work

5.1 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 [34] from a SCOOOL 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 SCOOOL’s. It has several interesting goals, some of them being common with ours:

Completeness It represent all standard C++ constructs.

Generality It is not targeted to a small area of applications.

Regularity It aims at having rules as general as possible rather than special cases.

Emphasis on types Types carries the semantics of a program.

Compiler neutrality The only reference should be the C++ ISO/IEC Standard [1].

Efficiency and elegance The tool should be able to process industry-level (and size) code base.

As SCOOOL is specially dedicated to the static world, it is not concerned by the first constraint: some C++ constructs, such as classic inheritance, are not represented. The other points match SCOOOL’s objectives. Unfortunately THE PIVOT is not available yet, and there is no documentation nor papers describing XPR and its usage.

5.2 Metagene

“ 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://projects.lrde.epita.fr/MetaGene>

Though the METAGENE [21] project was discontinued, a lot of interesting work was achieved and some of the developed ideas have been implemented in SCOOL. For example, seeing parameterized classes as static functions is a very interesting and powerful SCOOL feature that was introduced in METAGENE. The major difference with SCOOL, aside from the syntax, is that it is a language manipulating C++ code rather than being a representation of it. METAGENE allows the definition of functions such as these:

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

SCOOL aims at being a direct representation of C++ code; it not only features a neater syntax but also a more powerful and expressive language giving easily access to features which would require much more equivalent C++ code.

Metaprogramming METAGENE is handy to write C++ metaprograms. For instance, writing the power metafunction in METAGENE is quite natural; consider the following code, in an ML-like syntax.

```
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.

```
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 take only static parameters and return a type. Though they have not been implemented in METAGENE, they have been discussed and would have used the syntax illustrated by the following example.

```
let vector T = <c@ /* no base class */ @
public:
T& operator [](unsigned i)
{ return _data[i]; }
private:
T* _data;
@>
```

The concept of static functions is very powerful; it provides all the usual power of functions (partial or lazy evaluation, overloading, etc.) with a simple and natural syntax.

5.3 D

“ 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>

Traits Compared to C++, D provides more direct means to query information on types at compile time. For example, the traits system allows one to access to information such as whether a class is abstract (`isAbstractClass`), the nature of a function (`isVirtualFunction`, `isAbstractFunction`), the members of a class (`hasMember`, `getMember`), etc.

```
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: the user cannot define their own traits.

Metaprogramming As in C++, metaprogramming in D is performed thanks to complex uses of templates, though in the case of C++ the discovery of these techniques was more an accident than a formalized ability. This implies some

weird behaviors and very complicated code. The D language has been designed while keeping in mind those techniques, so as 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? ”

<http://www.digitalmars.com/d/2.0/templates-revisited.html>

Compile-time functions The D language provides the ability to execute usual functions at compile time when possible. There are obviously some constraints on a function so that it can be executed at compile time. A non-comprehensive list includes the following limitations.

- 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.

The following example illustrates this ability. Consider the **power** function that is either executed at compile time or at run time depending on the usage context.

```
int power(int x, int n) // Compute x^n.
{
    if (n == 0) return 1;
    else if (n == 1) return x;
    else return x * power(x, n - 1);
}

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

```

Static if The D language provides the ability to execute **static ifs**. This means that such tests are evaluated at compile time and may depend on types. SCOOOL's **static if** construct is similar to D's. The following example illustrates the use of the **static if** statement and template specializations in D.

```
// Return the type of the T * U product.
class product_trait(T, U)
{
    static if (is(T == U)) {
        typedef T res_type;
    }
    else static if (is(T == double) || is(U == double)) {
        typedef double res_type;
    }
    else static if (is(T == float) || is(U == float)) {
        typedef float res_type;
    }
    else static if (is(T == long) || is(U == long)) {
        typedef float res_type;
    }
    else {
        // Default return type.
        typedef double res_type;
    }
}

product_trait!(T, U).res_type product(T, U)(T t, U u) {
    alias product_trait!(T, U).res_type res_type;
    return cast(res_type)(t * u);
}

void main () {
    product_trait!(int, float).res_type a = product(42, 51.0f);
}
```

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

5.4 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 meta-programming 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

NEMERLE provides a powerful meta-programming system thanks to advanced, syntactic macros. Those are much more powerful than the lexical macros from the C and C++ languages. They are statically type-checked, may contain static statements, and can even extend the syntax of the language. The following example shows a **static if** as a macro in action.

```

using Nemerle.Compiler.Parsertree;

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

```

This other example shows how to add a C-like for loop:

```

macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")", body)
{
    <[
        $init;
        def loop () : void {
            if ($cond) { $body; $change; loop() }
            else ()
        };
        loop ()
    ]>
}

```

6 Conclusion

We have presented SCOOL, a new Domain Specific Language (DSL) for designers of efficient and generic C++ software in general, and scientific libraries in particular. SCOOL proposes a syntax simpler than C++'s for constructs usually hard to express in a few statements. It retains compatibility with C++, as it can embed C++ code within its definitions. SCOOL targets C++, therefore SCOOL users and C++ users should be able to work together and integrate their work seamlessly.

We report work in progress, as SCOOL is currently used only as a prototype in the OLENA project. Many OLENA contributors are not yet at ease with SCOOL for several reasons. First of all, SCOOL is a young project (about two years of development from a 1-2 developer team), that has not yet captured all of the idioms of an almost 10 year old project like OLENA. Secondly, the SCOOP paradigm did not reach maturity until recently, and SCOOL designers had to keep up with new ideas while SCOOP evolved. SCOOP now comes in two flavors. The first is the full-fledged paradigm (SCOOP 2 [10]), allowing complex designs thanks to abstractions reified as C++ classes, akin to C++0x's concepts. They are nevertheless more powerful as they can be used not only to constrain interfaces, but also to inherit from existing implementations, based on semantic properties attached to conforming types. The second version is a simplified one (informally called SCOOP 1.5), used within the current MILENA library, where some features have been sacrificed mainly for the sake of compilation times. Last, in spite of the high qualities of the STRATEGO/XT platform, learning a

new programming language (namely STRATEGO) is not easy. SCOOOL requires its contributors to first acquire some knowledge about third-party tools before actually taking part to the project. From the authors' point of view, despite the price of this admission ticket, the choice is worth it in the future, as the tools are definitely tailored to the task.

Future work on the SCOOOL translator includes adding a type-checker, supporting more transformations and developing a back end for C++0x, in particular to take advantage of reified concepts.

SCOOOL has produced working tools in one to two years, before the ones we expected from the five year old TRANSFORMERS project, mainly because the former has reduced the problem to a small subset of the goals of the latter. Still, now that the OLENA project has stabilized its core features, we expect a rapid coverage from SCOOOL, so that future OLENA's users can choose to work with SCOOOL or plain C++.

Acknowledgments Thierry Géraud discovered SCOOOL and initiated the SCOOOL project, initially implemented by Thomas Moulard and Quentin Hocquet. Valentin David and Nicolas Pierron provided invaluable help on STRATEGO/XT. Insight on NEMERLE was provided by Laurent Le Brun, while Ugo Jardonnet and Alexandre Fournier helped on D.

References

1. ISO/IEC: ISO/IEC 14882:2003 (e). Programming languages — C++ (2003)
2. Stepanov, A., Lee, M., Musser, D.: The C++ Standard Template Library. Prentice-Hall (2000)
3. The Boost Project: Boost C++ libraries. <http://www.boost.org/> (2008)
4. Siek, J.G., Lumsdaine, A.: The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In: International Symposium on Computing in Object-Oriented Parallel Environments. Number 1505 in Lecture Notes in Computer Science (1998) 59–70
5. The CGAL Project: CGAL, Computational Geometry Algorithms Library (2008) <http://www.cgal.org>.
6. Liskov, B., Zilles, S.: Programming with abstract data types. In: Proceedings of the ACM SIGPLAN symposium on Very high level languages, New York, NY, USA, ACM (1974) 50–59
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY (1995)
8. Veldhuizen, T.L., Jernigan, M.E.: Will C++ be faster than Fortran? In: Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97). Lecture Notes in Computer Science, Berlin, Heidelberg, New York, Tokyo, Springer-Verlag (1997)
9. The Boost Project: Generic programming techniques. http://www.boost.org/community/generic_programming.html (2008)

10. Géraud, Th., Levillain, R.: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In: Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'08), Paphos, Cyprus (July 2008)
11. EPITA Research and Developpement Laboratory (LRDE): The Olena image processing library. <http://olena.lrde.epita.fr> (2003)
12. Borghi, A., David, V., Demaille, A.: C-Transformers — A framework to write C program transformations. ACM Crossroads **12**(3) (Spring 2006) <http://www.acm.org/crossroads/xrds12-3/contractc.html>.
13. David, V., Demaille, A., Gournet, O.: Attribute grammars for modular disambiguation. In: Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06), Technical University of Cluj-Napoca, Romania (September 2006)
14. Moulard, T.: An overview of Scoop, a static object-oriented paradigm. Technical report, EPITA Research and Development Laboratory (LRDE) (2007)
15. van Noppen, M.: SCOOL: Concept-oriented programming. Technical report, EPITA Research and Development Laboratory (LRDE) (2008)
16. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16. Components for transformation systems. In: ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06), Charleston, South Carolina, ACM SIGPLAN (January 2006)
17. Lombardy, S., Poss, R., Régis-Gianas, Y., Sakarovitch, J.: Introducing Vaucanson. In Springer-Verlag, ed.: Proceedings of Implementation and Application of Automata, 8th International Conference (CIAA). Volume 2759 of Lecture Notes in Computer Science Series., Santa Barbara, CA, USA (July 2003) 96–107
18. Duret-Lutz, A.: Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In: Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—Young Researchers Workshop; published in “Net.ObjectDays2000”, Erfurt, Germany (October 2000) 653–659
19. Burrus, N., Duret-Lutz, A., Géraud, Th., Lesage, D., Poss, R.: A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In: Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL), Anaheim, CA, USA (October 2003)
20. Coplien, J. In: A Curiously Recurring Template Pattern. In [35].
21. Maes, F.: Metagene, a C++ meta-program generation tool. In: Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL; in conjunction with ECOOP), Oslo, Norway (June 2004)
22. Stroustrup, B.: A rationale for semantically enhanced library languages. In: Proceedings of the Workshop on Library-Centric Software Design (LCSD), San Diego, California, USA (October 2005)
23. Borghi, A., David, V., Demaille, A., Gournet, O.: Implementing attributes in SDF (May 2005) Communication to Stratego Users Day 2005.
24. Demaille, A., Largillier, Th., Pouillard, N.: ESDF: A proposal for a more flexible SDF handling (May 2005) Communication to Stratego Users Day 2005.
25. David, V., Demaille, A., Durlin, R., Gournet, O.: C/C++ disambiguation using attribute grammars (May 2005) Communication to Stratego Users Day 2005.
26. Bagge, O.S., Kalleberg, K.T., Haveraaen, M., Visser, E.: Design of the Code-Boost transformation system for domain-specific optimisation of C++ programs.

- In Binkley, D., Tonella, P., eds.: Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03), Amsterdam, The Netherlands, IEEE Computer Society Press (September 2003) 65–74
27. ISO/IEC: Working draft, standard for programming language C++. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2606.pdf> (May 2008)
 28. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: Proceedings of the First Workshop on C++ Template Programming, Erfurt, Germany (October 2000)
 29. Géraud, Th.: Advanced static object-oriented programming features: A sequel to SCOOP. <http://www.lrde.epita.fr/people/theo/pub/olena/olena-06-jan.pdf> (January 2006)
 30. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (July 1997)
 31. Visser, E.: A family of syntax definition formalisms. In van den Brand, M.G.J., et al., eds.: ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications, Technical Report P9504, Programming Research Group, University of Amsterdam (May 1995) 89–126
 32. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming* (2008) (To appear).
 33. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae* **69**(1–2) (2006) 123–178
 34. Stroustrup, B.: Elegant and efficient code. <http://parasol.tamu.edu/pivot/presentations/IAPO4.pdf> (May 2004)
 35. Lippman, S.B., ed.: C++ Gems. Cambridge Press University & Sigs Books (1998)