The preliminary investigation of template with C++

Shruti Sandal[1], Raghuraj Singh[2], Abdul Jabbar Khilji[3], Shashi Shekhar Ranga[4], Sanjay tejasvee[5], Devendra Gahlot[6]

[1,3,4,5,6]Department Of Computer Application
Engineering College Bikaner
2Head of Department of Computer Science & Engineering Department,
H.B.T.I., Kanpur
#1shrutisandal@yahoo.com,#3khiljisania746@gmail.com,#4ranga.ssr@gmail.com,
#5sanjaytejasvee@gmail.com, #6devendragahlot@gmail
*raghurajsingh@rediffmail.com

Abstract:- This paper describe the relationship between C++ templates and partial evaluation. In C++ ,templates were designed to support generic programming, but not deliberately provided the ability to perform compile-time computations and code generation. These features are completely deliberate, and as a result their syntax is ill at ease. After a review, these features in terms of partial evaluation, a much simpler syntax can be achieved. In C++, it may be regarded as a two-level language in which types are first-class values. Template instantiation resembles an offline partial assessor. In this paper, we explain groundwork in the direction of a single mechanism based on Partial Evaluation which unifies generic programming, compile-time computation and code generation. The language Catat is introduced to demonstrate these ideas.

Key Word :- Traits, Catat

I. INTRODUCTION

Templates were added to the C++ language to support generic programming. However, their addition by chance introduced powerful mechanisms for compile-time These mechanisms have proven themselves very useful in generating optimized code for scientific computation and code generation. computing applications [2,3,4,5]. Since they are accidental features, their syntax is somewhat awkward. The goal of this paper is to achieve a simpler syntax by recasting these features as partial evaluation. We start by briefly summarizing the capabilities provided by C++ templates, both intended and accidental.

(a) Template in C++:-Templates are a powerful but poorly understood feature of the C++ language. Their syntax resembles the parameterized classes of other languages (e.g., of Java). But because C++ supports template specialization, their *semantics* is quite different from that of parameterized classes. Template specialization provides a Turing-complete sub-language within C++ that executes at compile-time. Programmers put this power to many uses. For example, templates are a popular tool for writing program generators. The C++ Standard defines the semantics of templates using natural language, so it is prone to misinterpretation.

(b) Review of generic programming

Generic programming is a methodology for creating highly reusable and efficient algorithms. Language feature for writing some classes of polymorphic functions and data structure have received more attention than sound programming technique at the foundation of generic libraries. The creative goal of templates was to support generic programming, which can be summarized as “reuse through parameterization”. Generic functions and objects have parameters which modify their behavior. These parameters must be known at compile time. Functions may also be templates. Here is a function template which sums the elements of an array:

```
template<type name T>
T sum(T* array, int numElements)
{
T result = 0;
for (int i=0; i < numElements; ++i)
result += array[i];
return result;
}
```

This function works for built-in types, such as int and float, and also for user-defined types provided they have appropriate operators (=, +=) defined. Templates allow programmers to develop classes and functions which are very customizable, yet retain the efficiency of statically configured code[1].

(c)Computation in template at compile time:-Templates can be exploited to perform computations at compile time. This was discovered by Erwin Unruh [10], who wrote a program which produced these errors at compile time:

```
erwin.cpp 10: Cannot convert ’enum’ to ’D<2>’
erwin.cpp 10: Cannot convert ’enum’ to ’D<3>’
erwin.cpp 10: Cannot convert ’enum’ to ’D<5>’
erwin.cpp 10: Cannot convert ’enum’ to ’D<7>’
erwin.cpp 10: Cannot convert ’enum’ to D<11>’
```

The program tricked the compiler into calculating a list of prime numbers! This capability was quite accidental, but has turned out to be very useful.

(d) Code generation:-It turns out that compile-time versions of flow control structures (loops, if/else, case switches) can all be implemented in terms of templates.. These compile-time programs can perform code generation by selectively in lining code as they are “interpreted” by the compiler. This technique

CP1324, *International Conference on Methods and Models in Science and Technology (ICM2ST-10)*
edited by R. B. Patel and B. P. Singh

is called template metaprogramming [7]. Here is a template metaprogram which generates a specialized dot product algorithm:

*(e)Techinque to define function-Traits:-*The traits technique [6]allows programmers to define "functions" which operate on and return types rather than data. For ex.-If the array contains integers, a floating-point result should be returned. But a floating-point return type obviously will not suffice for a complex-valued array. The solution is to define a traits class which maps from the type of the array elements to a type suitable for containing their average.

II PARTIAL EVALUATION OF DATA BY TEMPLATE

Partial evaluators [8]regard a program's data as containing two subsets: static data, which is known at compile time, and dynamic data, which is not known until run time. A partial evaluator evaluates as much of a program as possible (using the static data) and outputs a specialized residual program. To determine which portions of a program may be evaluated, a partial evaluator performs binding time analysis to label language constructs and data as static or dynamic. Such a labeled language is called a two-level language.

*(a) C++ as a two-level language:-*C++ templates resemble a two-level language. Function templates take both template parameters (which have static binding) and function arguments (which have dynamic inding).

(b)Off line partial Evaluation:- Partial evaluation of languages which contain binding-time information is called offline partial evaluation. Template instantiation resembles offline partial evaluation: the compiler takes template code (a two-level language) and evaluates those portions of the template which involve template parameters (statically bound values).

*(c) Catat :multi-level language based on C++:-*Here we discuss preliminary ideas for a single mechanism based on Partial Evaluation which unifies generic programming, compile-time computation, and code generation. To demonstrate the ideas, we pioneer a (currently hypothetical) language Catat. Catat is a multi-level language based on C++ in which types are first-class values.

III FEATURE OF CATAT

*(a) Function:-*Functions in Catat may take a mixture of static and dynamic arguments. We find it convenient to give functions two separate parameters lists, as in C++. Here is an implementation of the meta dot function described earlier:

```
function dot(int@ N, type name@ T)(T* a, T* b)
T result = 0;
for@ (int@ i=0; i < N; ++i)
result += a[i]*b[i];
return result;
}
```

The concept is easier to express in a functional notation:

```
(define dot
(lambda (static-parms)
(PE static-parms
(lambda (dynamic-parms)
body))))
```

where (PE parms expr) performs partial evaluation of expr using static parameters parms. The use of argument lists of the form (static-parms)(dynamic-parms) hints at this idea, and also avoids the parsing difficulty associated with <> brackets in C++. Catat discards the return type specification of C++ and replaces it with the keyword function. The return type may result from compile-time calculations, and so must be inferred from the body of the function. They are not fixed to any stage.

*(b) Specialization:-*When calls to function templates are encountered during C++ compilation, the template is instantiated. In Catat a similar process would occur, which may be called specialization: a partial evaluator produces a residual function by evaluating the static constructs. This function call:

```
int data[10];        // ..
float result = average(int)(data,10);
```

triggers the partial evaluation of average; the resulting specialization (translated into C++) might be

```
float average__int(int* array, int N){
float sum = 0;
for (int i=0; i < N; ++i)
sum += array[i];
return sum;
}
```

*(c)Binding time specifications :-*Each scope in a Catat program is associated with a default binding time. By default, the global scope has dynamic binding. To indicate statically bound variables, an @ symbol is appended to the type:

```
int i = 0;          // Dynamic data
int@ j = 0;      // Static data
```

The type int@ is equivalent to const int in C++.

To preserve consistency between the dynamic and static versions of the language, it is necessary to allow multiple levels of binding (or stages). The @symbol indicates that a variable is bound in the previous stage. The @ symbol may also be applied to control constructs:

```
// Calculate N! (factorial) at compile time
int@ N = 5, Nfact = 1;
for@ (int@ i=1; i < N; ++i)
Nfact *= i;
```

Operators such as=and*are applied at compile-time if their operands are statically bound. Data may flow from static to dynamic constructs, but not vice-versa. This is called cross-stage persistence by Taha and Sheard [11]. For example:

```
{
int@ i;
int j;
j = i;                    // Okay, i is known at runtime
i = j;                    // Not okay, j not known at ctime
```

(d)Compling by catat

To compile Catat as described, one apparently needs both a Catat-interpreter and a Catat-compiler. The following steps should be taken:

1.Use the interpreter to partially evaluate

2.Use the compiler to produce native code for the residual

function.

It may be possible to avoid this problem by using an approach similar to that pioneered by the Cmix partial evaluation system [1]. The basic approach is to use a "closure compiler" which uses run-time code generation (RTCG) to compile a single function. RTCG is a bit of a misnomer, since the code generation is being done at compile-time by the compiler.

IV. MOTIVATING POTENTIAL

There are some motivating Potential which increases when they are uses with some language like catat.

(a) Scripting: The partial evaluator for Catat needs to contain what is essentially an interpreter to evaluate the static portions of the program. This implies that you get scripting for no extra cost; a Catat program consisting solely of static constructs will be completely interpreted, with no residual code generated.

(b) Reflection and Meta-level Processing A language like Catat may provide a natural environment for implementing reflection and meta-level processing capabilities, since the ability to perform compile-time calculations is there already. Such capabilities would allow programmers to query objects about their methods and members, determine the parameter types of functions, and perhaps even manipulate and generate abstract syntax trees.

V. ALLIED WORK OF LANGUAGE

Nielson and Nielson [15] first investigated two-level languages and showed that binding- time analysis can be expressed as a form of type checking. The most closely related work is MetaML, a statically typed multi-level language for hand-writing code generators [11]. MetaML does not appear to address the issue of generic programming. Gluck and Jørgensen described a program generator for multi-level specialization [12]which uses a multi-level functional language to represent automatically produced program generators. Metalevel processing systems address many of the same problems as Catat; they give library writers the ability to directly manipulate abstract syntax trees at compile time. Relevant examples are Xroma [13], MPC++ [14], Open C++ [15], and Magik [16].These systems are not phrased in terms of partial evaluation or two-level languages; code generation is generally done by constructing abstract syntax trees. A more closely related system is Catacomb [17], which provides a two-level language for generating runtime library code for parallelizing compilers.

VI.CONCLUSIONS

In this paper we discuss that C++ with template may be as two level language and also static binding. Second is template instantiation bears a striking Languages build may offer a way to provide generic programming, code generation, and compile-time computation via a single mechanism with simple syntax. Similarity to offline partial evaluation.

REFERENCES

[1] C++ Templates as Partial Evaluation.

[2] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, andT. Williams, Array design and expression evaluation in Lecture Notes in Computer Science.

[3] J. G. Siek and A. Lumsdaine, The Matrix Template Library: A generic programming approach to high performance numerical linear algebra, in International Symposium on Computing in Object-Oriented Parallel Environments, 1998.

[4] Using C++ template metaprograms, C++ Report, 7 (1995), pp. 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.,Arrays in Blitz++, in ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science,1998.

[5] T. L. Veldhuizen and K. Ponnambalam, Linear algebra with C++ template metaprograms, Dr.Dobb's Journal of Software Tools, 21 (1996), pp. 38–44.

[6] N. Myers, A new and useful template technique: "Traits", C++ Report, 7 (1995), pp. 32–35.

[7] Using C++ template metaprograms, C++ Report, 7 (1995), pp. 36–43. Reprinted inC++ Gems, ed. Stanley Lippman.

[8] N. D. Jones, An introduction to partial evaluation, ACM Computing Surveys, 28 (1996), pp. 480–503.

[9] Y. Futamura, Partial evaluation of computation process - an approach to a compiler-compiler, Systems, Computers, Controls, 2 (1971), pp. 45–50.

[10] E. Unruh, Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462.

[11] W. Taha and T. Sheard, Multi-stage programming with explicit annotations, ACM SIGPLAN Notices, 32 (1997), pp. 203–217.

[12] R. Gl¨uck and J. Jørgensen, An automatic program generator for multi-level specialization, Lisp and Symbolic Computation, 10 (1997), pp. 113–158.

[13] K. Czarnecki, U. Eisenecker, R. Gl¨uck, D. Vandevoorde, and T. L. Veldhuizen, Generative Programming and Active Libraries, in Proceedings of the 1998 Dagstuhl-Seminar on Generic

[14] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota, Design and implementation of metalevel architecture in C++ – MPC++ approach, in Reflection'96, 1996.

[15] S. Chiba, A Metaobject Protocol for C++, in OOPSLA'95, 1995, pp. 285–299.

[16] Incorporating application semantics and control intoompilation, in USENIX Conference