Artem Pelenitsyn <ulysses4ever@gmail.com>

WGP 2012 notification for paper 6

WGP 2012 <wgp2012@easychair.org> 27 июня 2012 г., 23:46
Кому: Artem Pelenitsyn <apel@sfedu.ru>

Dear Artem Pelenitsyn,

We regret to inform you that your paper

Generic Programming Approach in the Implementation of Error-Correcting Codes From Algebraic Geometry

was not selected by the program committee to appear at the
Workshop on Generic Programming 2012. Nevertheless, we hope
you will be able to attend the workshop.

We have enclosed the reviewers' comments for your perusal.
we hope they will offer you guidance in revising or
re-targeting your paper. If you have any additional questions,
please feel free to get in touch.

Thanks for submitting to WGP 2012.

Andres Loeh and Ronald Garcia -- Program Co-Chairs

----------------------- REVIEW 1 ---------------------
PAPER: 6
TITLE: Generic Programming Approach in the Implementation of Error-Correcting Codes From Algebraic Geometry
AUTHORS: Artem Pelenitsyn

OVERALL RATING: 1 (weak accept)
REVIEWER'S CONFIDENCE: 2 (medium)

The paper describes a generic library that implements the "BMS" algorithm for error-correcting codes based on algebraic geometry.

Overall, the paper is right on top for WGP: it applies modern C++ generic programming techniques to a new domain, providing implementation experience. My main criticism of the paper is that it focuses on the wrong details of the implementation: the techniques that it discusses in some depth (traits, policy classes, enable_if, etc.) are well-known techniques within the C++ community. While the expositions are good, they aren't the most interesting part of this work, yet they consume most of the space in the paper.

The most interesting part of this work is in the conceptualization of the domain of error-correcting codes based on algebraic geometry. The author hints in 1.3 at the concepts one must use to provide one's one data type for the library. I would love for the paper to describe those concepts in depth so they can help the reader better understand the domain of ECC and how those concepts interact with (and are determined by?) the BMS algorithm. Such concepts also provide the foundation that allows the author's work to be tied in with other generic libraries in other domains, because concepts often translate, which makes it even more important to discuss in this paper.

A few minor comments:
- at the end of an introduction, a very rough roadmap would be useful, but you don't need to tell us how

many sections there are. Impatient readers can scan ahead if they want.

- In section 1, please provide more of an introduction to the field of error-correcting codes here. This paper is likely to be of interest to people who understand generic programming but have never dealt with error-correcting codes, and therefore need some introduction to this domain. For example, what is the BMS algorithm? How (generally) does it work?

- The assumption that the C++11 code will compile with Visual C++ because it compiles with GCC 4.6 isn't likely to be true, because Visual C++ has far less of C++11 implemented than GCC 4.6. If you want to say something about adherance to the C++11 standard, I suggest compiling with Clang 3.1; if you want to say something about portability to Windows, test with Visual C++. Or, you could simply remove this paragraph (which is easiest): it doesn't really add anything to be paper.

- In section 2.3, the call inc(data) is not actually correct due to two-phase name lookup. See http://clang.llvm.org/compatibility.html#dep_lookup_bases for information on two-phase name lookup in dependent base classes.

- In section 2.4: I found it odd that you first mention and then use lambdas, then also use std::bind (which lambdas typically replace). There's also no point in mentioning the import of std::placeholders; more important, if you keep the use of bind, would be to say what _1 actually does. More importantly, however, this is a really important section in the talk: I'd much rather that you focused on what the algorithm is doing (and on what concepts are important for this algorithm to operate correctly and efficiently) rather than discuss C++11 details.

- Measuring performance of the library with various instantiations (e.g., for the four NTL types mentioned) would be very helpful, to demonstrate that the algorithm generalizes to these various types. It's not critical for the paper, but it helps give the reader a sense of what matters for the performance of this algorithm. Ideally, if there are other (non-generic) implementations freely available, it would be wonderful to compare performance against those implementations as well.

In general, I would love to see the C++ implementation details receive less emphasis in the paper, and instead enrich the discussion of the underlying ECC domain, including more discussion of the BMS algorithm and how it interacts with the concepts in the library.

Also, what does the future hold? Do these concepts translate to other ECC algorithms, such that one could build a library? Are there other related domains that this library could interact with?

Will you be making the implementation available to others? I would love to see it generally available, so others can learn from it.

----------------------- REVIEW 2 ---------------------
PAPER: 6
TITLE: Generic Programming Approach in the Implementation of Error-Correcting Codes From Algebraic Geometry
AUTHORS: Artem Pelenitsyn

OVERALL RATING: -3 (strong reject)
REVIEWER'S CONFIDENCE: 4 (expert)

This paper presents an implementation of error-correcting codes, mostly based on C++ templates, and in particular highlights the use of certain features of C++11.

As is well-known, mathematics is an extremely fertile ground for generic programs of all sorts. There are whole languages (Axiom, Aldor) whose very purpose in life is to leverage genericity in mathematics; these have also had a non-trivial influence on the design of other languages such as Magma. Going further back in time, Musser and Stepanov (frequently in collaboration with Kapur) worked on Tecton, influenced by their earlier work in computer algebra, which eventually led to the design of the STL for C++. Of course, from there a rich tradition evolved, introducing traits and concepts as ways to organize generic programs. This led to quite a few implementations of C++ template libraries, many focusing on mathematics -- not all of which are examined/cited here. In particular, LinBox, POOMA and CGAL come to mind.

All of this is relevant to the current paper, although a lot of the groundbreaking work on these topics is not cited. The paper aims to present some design decisions about a particular implementation. But the paper is really a description of an implementation, and then a discussion of some of the implementation decisions. Usually when one describes software, especially design decisions, it is customary to first define the problem domain (in a software-independent manner), then outline more detailed requirements, before leading into design decisions. Lower level implementation details are only described if they really matter. None of this higher level material is present in this paper.

Furthermore the design decisions highlighted here are seriously outdated. Anyone with the least bit of knowledge of symbolic computation would know that although Q[x,y] is isomorphic to (Q[x])[y] (and (Q[y])[x]), from an efficiency perspective these choices matter tremendously, especially as the number of variables goes up. Also, proper generic representations usually allow both dense and sparse representations, as in practice most multivariate polynomials are sparse. Another item which is surprising is that more mathematical structures (rings, fields, monoids) are not important in the implementation. They are clearly there (implicitly!) in the fragments presented.

From a data-generic perspective, some of presented code is also quite puzzling: operator*= on p.3 is just 'map (* c)' (in Haskell notation) -- and very long and complex definition thereof. What is less obvious is that this implements a scalar right-action from the scalar field onto the polynomials -- while normally we speak of left-actions. Of course this does not matter in this case as the underlying multiplication is commutative. But these are extremely important issues to discuss in the context of 'generic' programming!

Apart from these fundamental errors, I simply cannot tell what new principles are to be learned from here. Fundamentally, I do not think there are any -- all the design principles needed are either already described in the C++ articles cited here, or in Computer Algebra books which ought to be cited. Worse, some of these pre-existing principles have been mis-applied, leading to a sub-optimal implementation.

---------------------- REVIEW 3 --------------------
PAPER: 6
TITLE: Generic Programming Approach in the Implementation of Error-Correcting Codes From Algebraic Geometry
AUTHORS: Artem Pelenitsyn

OVERALL RATING: 1 (weak accept)
REVIEWER'S CONFIDENCE: 2 (medium)

The author provides the description of his generic library to implement a decoder for a class of algebraic geometry codes. The library implementation utilize a number of generic programming and C++ metaprogramming techniques.

This is a well-written workshop paper: presents a library idea,
implementation details and the applied techniques. My only concern is
that the purpose and the actual usage of the library is not explained
in the level that the reader who is not familiar with the actual area
can fully understand it. It is also clear that the library has not been
tested on many platforms as well as no performance comparisons exists yet.

A historical remark:
On page 4 7th par: typelists have been mentioned first in:
K. Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools,
and Applications (Addison-Wesley Longman, 2000).