Teaching Compiler Development using .NET Platform

Andrey Terekhov
Microsoft Russia & CIS
Chapaevsky per., 14
125252, Moscow, Russia
terekhov@acm.org

Dmitry Boulychev, Anton Moscal, Natalia Voyakovskaya
St. Petersburg State University
Mathematical & Mechanical department
Universitetsky pr., 28, room 2360
198504, St. Petersburg, Russia
{db, msk, nat}@tercom.ru

ABSTRACT

We present our experience of teaching in the universities compiler development on the basis of .NET platform. We discuss typical problems of teaching compiler development and our approach to dealing with these problems. We consider applicability of .NET/Rotor in this context and share the lessons learned during preparation and delivery of this academic course.

Keywords

Compilers, teaching, .NET, Shared Source CLI, Rotor

1. INTRODUCTION

Compiler development is one of the oldest and the best researched topics in software engineering. It is a fundamental part of the universities computer science curricula. However, it is also one of the most difficult topics to teach. Students often find courses on compilers hard, because they have complex theoretical foundation and exercises require tedious coding. In most cases, the size of compilers that students have to write during the course exceeds anything they produced earlier. For these reasons, development of a course in compilers merits special consideration. The goal is to support early interest and understanding of the subject, and retain students' motivation throughout the course.

In 2001 we started rewriting an existing academic course on compiler development which ran in St. Petersburg State University since early 1970s. This course is offered to the students in the 3rd year of education and lasts for one academic semester (four calendar months). The course has been regularly updated every 5 to 10 years. At the beginning of our project it was based on the architecture of Intel microprocessors. At that time we have already had an experience of working with early releases of Visual Studio .NET and came to a conclusion that .NET represents a future-proof platform that could be used as a basis for the course on compiler development.

In March 2002 Microsoft announced its Shared Source Initiative (see http://sharedsourcecli.sscli.net), an open-source implementation of .NET that was informally code-named Rotor, and we launched a brief investigation on whether Rotor would be a better fit for the purposes of our course. It turned out that Rotor provides students with an excellent opportunity to become acquainted with a real-life compiler, as well as to get experience of working with the large (3.5+ million lines of code!) software code base while still at the university.

Simultaneously with Rotor's announcement, Microsoft Research issued a Request For Proposals aimed at supporting Rotor-based research and education projects. Our group was awarded one of the grants under this initiative. We have created a complete set of presentations and lecture notes for one-semester academic course on "Compiler Development for .NET Platform" in Russian and English languages, which is available at the Web-site of St. Petersburg State University (see http://www.iti.spbu.ru/eng/grants/Cflat.asp).

In this article we present the experience gained during preparation and delivery of this academic course. The article is organized as follows. In Section 2 we briefly discuss advantages and disadvantages of .NET platform from the point of view of supporting various programming languages. Section 3 presents typical problems of teaching compiler development and our approach to handling them. Section 4 contains a general overview of the course and a description of the deliverables that we have created during this project. In Section 5 we illustrate our approach to teaching compiler development using excerpts of the lectures. Finally, Section 6 summarizes our experiences so far and outlines some directions for further research.

2. WHY .NET?

One of the more popular directions of the last decade is *virtual machines* – a powerful concept, which abstracts away the differences between hardware platforms and thus enables portability of programs written in a particular programming language.

At the moment .NET is arguably the most promising of those virtual machines, because it was designed from the very beginning to support most of the existing programming languages, unlike previous efforts that were aimed at a single language. Thus .NET is a convenient platform for compiler development, which is more powerful than its predecessors, such as Java:

- In .NET there exists a special API for code generation (Reflection.Emit), while Java provides only file generation methods
- In .NET it is possible to pass a reference as a parameter and as an output value (in C# this options are represented by keywords **ref** and **out**). To emulate such behavior in Java one has to create a wrapper class that would be placed in a heap.
- .NET platform provides support for important encapsulation and abstraction mechanisms, such as properties and indexers. In Java this cannot be implemented directly, so one has to settle for the use of naming conventions, which the compiler does not verify.
- In .NET it is possible to generate unsafe (i.e., unverifiable) code. This could be useful, for instance, for the purposes of achieving runtime efficiency or integration of legacy systems. In Java this is possible only by calling programs written in other languages, such as C.

Naturally, .NET is not the perfect solution, and some of its advantages are based on subtle design trade-offs, which are especially visible during implementation of languages that do not correspond directly to the .NET model. Here are some features typical for various programming languages, but difficult to implement in a compiler to .NET platform:

- Multiple inheritance (Eiffel, C++)
- Nested procedures (Pascal, Algol 68)
- Parametric polymorphism (ML, Haskell)
- Constructors with user-defined names other than the name of the class (Pascal)
- Non-standard data types (for instance, consider the problems of supporting PICTURE data type used in Cobol and PL/I)

Finally, writing a compiler from almost any functional languages to .NET is somewhat problematic, because .NET is heavily biased towards traditional imperative languages. Such a compiler would lead to inefficiency of the generated code or would require generating unverifiable code.

Nevertheless, practice has shown that these problems are not crucial – there already exist dozens of compilers from various languages to .NET, and new compilers keep appearing. Writing a compiler for .NET platform as an exercise is relevant for the students ..NET continues to evolve – some of the above mentioned problems are already obsolete and others will probably get resolved in the upcoming releases of .NET (see .NET version 1.2, and research projects such as Gyro, see http://research.microsoft.com/projects/clrgen and ILX, see http://research.microsoft.com/projects/ilx).

3. ISSUES OF TEACHING COMPILER DEVELOPMENT

The following technical and psycholiogical issues need to be considered in preparation and delivery of a compiler development course [Chanon75, Appelbe79].

First of all, compiler courses deal with the complexity of the problem domain, especially with the abundance of mathematical theory. This presents more difficulty for students majoring in software engineering, since their curriculum is usually more practical than theoretical. In some cases, this problem leads to over-emphasis on theory at the expense of practical usefulness of the course; in other cases the course becomes all-embracing and overly timeconsuming for students. As a result, there exists a gap between compilers "as taught in the universities" and compilers "as written in the industry".

In order to overcome this problem, we tried to minimize the amount of theory by describing only those formalisms and theorems that are directly required for understanding the material of the course. Nevertheless, our course includes introduction to language and grammar theory, automata theory, data and control flow analyses, so purely theoretical material constitutes about one third of our course.

We also found it useful to separate the text of the lecture notes into "main text" and "digressions"., Main text contains theoretical explanations and description of universally accepted practices of compiler construction, while digressions are the advanced topics, such as practical tricks that are useful only under certain conditions or could be employed to overcome various limitations of the straightforward approaches[1]. This is useful for structuring the theoretical material of the course and presents the student with two different perspectives on compiler writing, from the computer science and software engineering points of view. Experienced students may also take advantage of this separation by concentrating only on those parts that are less well-known for them.

Secondly, for most students the size of compiler that they have to produce is much greater than all their previous projects at the university. In order to be successful, courses on compilers should run in the interactive mode and contain many possibilities for the student to get clarifications. One of the main methods to achieve this is to complement the lectures with the self-paced independent work by students, which, in our opinion, should be organized at regular hours in the university computer labs and should be supervised by either lecturer or assistant.

This course tries to teach not only the basics of compiler writing (so called "programming-in-the-small"), but also issues that are important for working with large code base (so called "programming-in-the-large"). We try to achieve both of these goals by first demonstrating the concepts of compiler writing using examples taken from a demo compiler of a simple language C-flat, which is a subset of C#, and then by illustrating advanced topics using examples taken from a full-blown compiler of C# that is available in Rotor. This approach shows the student the whole set of "under-the-hood" details of compilers that are usually too complicated for implementation in "toy languages" and are quite often omitted in academic courses on compilers:

- Possibility to illustrate various platform-dependent aspects, such as run-time support and generation of debugging information
- Demonstration of garbage collection, JIT-compilation and other system mechanisms
- System and auxiliary tools (assembler, disassembler, debugger etc.)
- Implementation of Foundation Class Library classes

Note that one cannot guarantee that the algorithms used in Rotor are equivalent to those used in Visual Studio .NET, since their goals are different – Rotor is designed to be as clear and understandable as possible, while .NET is striving to achieve maximum efficiency of the generated code. This makes Rotor good for teaching, but creates a risk that students will mechanically imply that the same algorithms work in industrial implementation and will rely on that false assumption in their work, so the lecturer should explicitly draw students' attention on this difference.

Finally, for some of the students understanding the target platform may be difficult, especially, at the code generation phase. The knowledge of .NET platform is not yet widespread, so it was a real problem of our course. We recommend starting the course with a two-lecture overview of .NET platform presented from the programming point of view. However, it might be a better idea to consider knowledge of .NET as a pre-requisite for this course. We have already started transition to this model, because now there is a separate course on .NET available for the students on elective basis earlier in their studies in St. Petersburg State University. This course was devised by one of the authors, Andrey Terekhov, and based on a well-known book [Richter02]. We believe that in the foreseeable future .NET will gain more popularity both in industry and academia and thus more universities will view this approach as a better alternative.

4. OVERVIEW OF THE COURSE

As a result of preliminary research and planning we came up with the following requirements to our

[1] The idea of separating material into "main text" and "digressions" was traditional for mathematical textbooks of the Soviet era. Typically, digressions represented reading that was not required for the students and were typeset in fine print.

course – the course should unite both theory and practice, complimenting both the theory-oriented text books, such as [Aho86, Muchnick97], and practice-oriented books, such as [Gough02]. The course should be based on .NET and particularly on examples taken from its open implementation, Rotor.

We wrote lecture notes and slides for this course based on the above requirements. The phase of active development lasted for about a year. As a result of this activity, we created the following 15 lectures:

1. Overview of .NET and Rotor
2. Overview of C#
3. Compiler Basics
4. Language Theory
5. Lexical Analysis
6. Syntax Analysis – Recursive Descent
7. LR(k) and LALR Grammars
8. Grammars and YACC
9. Semantic Analysis. Internal Representation
10. Memory Management
11. Optimization
12. Control Flow Analysis
13. Data Flow Analysis
14. Generation of CIL
15. Instruction Selection during Code Generation

Note that some of the lectures require more time for delivery than the usual one and a half hours that are typically allotted in Russian academic system, so this list represents logical division of the course into related topics rather than the recommended duration. We assume also that during the semester the students will additionally spend a comparable amount of time on review of source codes of C-flat and Rotor, and will independently implement a sample compiler according to the individual tasks set out by the lecturer.

The course material includes a sources and binaies of ademo compiler of a "toy language" called C-flat. The grammar of this language is intentionally simple – BNF grammar of C-flat takes less than 30 lines. Anton Moscal, one of the authors of the course, has produced the C-flat compiler.The course refers often to the compiler sources. We are currently trying to bootstrap C-flat compiler, i.e. we are trying to rewrite C-flat compiler in C-flat. This is a dual process, which requires both changing the compiler (i.e., using less powerful language constructions) and changing the language (i.e., expanding the set of allowed constructions).

In 2002-03 we made a pilot delivery of some of our lectures to the students of St. Petersburg State University. The lectures were well-received by the audience and generated a lot of feedback that we used to improve the contents of the course.

We also proposed topics for term work on the basis of the course to the students. One of the goals of these term assignments was to assess validity of our assumptions about students' knowledge prior to the course and the difficulty of course material. For instance, a 3rd year student was given a task to implement a C-flat compiler in C# in order to make sure that it is possible for a student to develop such a compiler during one semester. In another term project, a team of 4th year students was asked to develop a compiler from subset of Pascal to .NET that would be written in SML.NET using MLLex/MLYacc. Both of these projects were successfully completed, which suggests a strong evidence of importance of this course and relevance of its content.

At the moment we are considering several ideas on further development of the course. One of the ways to improve the course is to enlarge the scope by adding material on Mono project (see http://www.go-mono.com). Mono is another open-source implementation of .NET, which is interesting due to the fact that it uses different approaches to implementation of various aspects of .NET than Rotor, for instance:

- Mono C# compiler is written in C# and thus capable of bootstrapping itself. It might be argued that it is also more readable from the student's point of view than Rotor's compiler written in C++
- Garbage collection is implemented using Boehm's conservative garbage collector for C [Boehm88, Boehm93], which is a radically different approach to solving memory management issues
- Unlike Rotor C# compiler, Mono uses BURG [Pelegrì-Llopart88, Aho89, Fraser92] approach for instruction selection. This topic is briefly mentioned in the last lecture of our course, and thus Mono presents a good opportunity to illustrate this theory with a real-life example

5. AN EXAMPLE OF LECTURE MATERIAL

To demonstrate some of the ideas behind this course, we will briefly walk through the material of the lecture on memory management and garbage collection. This lecture relies heavily on examples from Rotor source code that are mostly adopted from the book [Stutz03], which we recommend as a supplementary reading to students.

We start the lecture with asserting that memory management is an extremely important and resource-consuming problem in programming. According to a classical textbook [McConnell93] in typical C projects memory management consumes up to 50% of the time dedicated to coding and debugging. As an example we consider Rotor's C# compiler that is written in C++ and thus is based on manual memory tracking and disposal by the programmer. In order to perform this task correctly, developers of C# compiler had to come up with a number of auxiliary structures, for instance, below we enumerated all the places where Rotor stores information about an object instance:

- Memory for object instances is stored in a garbage-collected heap (excluding SyncBlock, which is stored inside the execution engine itself)
- Method table of the object is placed in the “frequently used” heap of its application domain; in the meantime EEClass, FieldDescs and MethodDescs of the object are placed in “rarely used” heap
- Native code, generated by JIT-compiler, is placed in the code heap and is shared by all application domains
- All other stuff related to object (for instance, stubs generated for this object) are stored in a separate memory region of the execution engine

Clearly, manual tracking of all these elements is a tedious task and thus modern programming languages and platforms tend to rely on automatic memory management instead.

Then we provide an overview of existing methods of memory management. The students should already have this knowledge from the course on fundamentals of programming languages, but we believe that it is always helpful to provide a brief refresher on this topic.

We proceed to a detailed discussion of garbage collection scheme that is used in Rotor. This is important for students because it enables them to connect theoretical description from the previous part of lecture with the concrete implementation. The main focus is on practical consequences of the theory that we have just discussed and on the fact that programmers always have to deal with tricks and heuristics in order to increase efficiency of the implementation.

First, we emphasize the general approach to garbage collection in Rotor and explain the reasoning behind choosing particular garbage collection methods. We mention that Rotor uses hybrid scheme of garbage collection with two generations:

- Generation 0 is compacted by copying — this pays off since the majority of the objects goe away before the first GC, so not a great deal of copying takes place
- Generation 1 is collected by mark & sweep — copying would not be advantageous here, since few objects are dying in this generation

There are many interesting details:

- Copying is always performed to a new heap (to be more precise, to a new segment of the heap, see below); this process is not overly expensive because it takes place separately in different generations
- Large objects are collected in a separate heap *without copying* (see below)
- Completely different scheme exists in Rotor for garbage collection of remote objects (see sscli/clr/src/bcl/system/runtime/remoting)

Note that the issue of GC for remote objects could be also discussed in a separate lecture “Remoting in .NET”, which should be a part of a separate academic course on .NET platform (this is the case for St. Petersburg State University, where these courses run in parallel).

Then we illustrate the implementation details by going from the top. We point to the main entry point of garbage collection – `GCHeap::GarbageCollect()`. We explain that it is called whenever a garbage collection is needed and enumerate possible scenarios for that to happen – for instance, if memory runs out, or an application domain is being unloaded, or finalizers have just completed, or if there was an explicit call by the programmer, during exit from the process and during debugging from the profiler. After the call to this function, all threads are suspended except the thread that performs the GC – this is achieved by call to the following function:

```
SuspendEE(GCHeap::SUSPEND_FOR_GC);
```

Note that prior to that all threads should reach their "GC safe" state (this is a good place to explain what are the characteristics of this state). Then the control is passed to the main function, **gc_heap::gc1()**. It works as follows – an attempt of copying garbage collection in the zero generation is made (as a result, all live objects are moved to the first generation). If garbage collection is required for the first generation as well (this is almost always the case), then mark-and-sweep is performed in it.

After these high-level explanations, we walk the students through the actual Rotor code that performs these tasks and explain the implementation details. This is quite easy to do, because the code itself is properly commented and readable, not even to mention the detailed explanations provided in the book [Stutz03]. For instance, below is the code that we use to illustrate the first stage of GC, `gc_heap::copy_phase()`:

- Live objects are found by recursive search and copied into the elder generation:

```
// Promote objects referred to by cross-generational pointers
    copy_through_cards_for_segments (copy_object_simple_const);
    copy_through_cards_for_large_objects (copy_object_simple_const);
// Promote objects found on the stack or in the handle table
    CNameSpace::GcScanRoots(GCHeap::Promote, condemned_gen_number,
max_generation, &sc, 0);
    CNameSpace::GcScanHandles(GCHeap::Promote, condemned_gen_number,
max_generation, &sc);
// Promote any object referred to from the finalization queue
    finalize_queue->GcScanRoots(GCHeap::Promote, heap_number, 0);
```

- References to these objects are updated:

```
// Relocating cross generation pointers
    copy_through_cards_for_segments (get_copied_object);
    copy_through_cards_for_large_objects (get_copied_object);
// Relocating objects on the stack or in the handle table
    CNameSpace::GcScanRoots (GCHeap::Relocate, condemned_gen_number,
max_generation, &sc);
   CNameSpace::GcScanHandles(GCHeap::Relocate, condemned_gen_number,
max_generation, &sc);
// Relocating finalization data
    finalize_queue->RelocateFinalizationData (condemned_gen_number, __this);
```

Table 1. Rotor code used to illustrate the copying stage of garbage collection

After discussion of garbage collection we also briefly mention *code pitching*. This is just one of the examples of the topics that are difficult for the students to grasp (it does not come easily to the students that not only program data, but also the generated code could be treated as garbage) and yet is easily demonstrated using the Rotor source code. As this is more or less an aside detail, we illustrate only the general idea of code pitching and leave the implementation details for students' independent study. The scheme of this part goes as follows:

- When the size of the heap for the compiled code exceeds some predefined maximum, the whole contents of the buffer is thrown away and all return addresses on the stack are replaced with address of thunk that causes re-compilation of methods
- To make a decision on throwing the code away, this process takes into account a lot of parameters (size of native code, ratio of native code to IL, time of JIT-compilation of the method etc.)
- See sscli/clr/src/vm/ejitmgr.cpp

We believe that it is important to make the students study the source code of real-life compilers, because it provides a much better way to learn how to write the code and acquaints the students with all the intricate details before they encounter them in their professional work after graduation.

6. ACKNOWLEDGEMENTS

This work was financially supported by the Rotor grant by Microsoft Research. We would also like to thank Vladimir Pavlov (eLine Software Inc.), Dmitry Malenko (Dnepropetrovsk National University, Ukraine) and anonymous reviewers for their valuable comments that helped to improve this paper.

7. CONCLUSIONS

We presented an academic course on compiler development that is based on .NET and Rotor. In this course we tried to bridge the gap between compilers "as taught in the universities" and "real-world compilers" by demonstrating both theoretical and practical perspectives on compiler development. From our point of view, .NET can be successfully used as a platform for education and research in various areas of computer science and software engineering, such as compilers, programming languages and component architectures. We also found out that Rotor is an especially interesting platform for education because it enables students to get acquainted with typical problems of working with industrial large-scale software projects.

8. REFERENCES

[Appelbe79] B. Appelbe "Teaching Compiler Development", In Proceedings of the 10th SIGCSE Technical Symposium on Computer Science Education, 1979, pp. 23-27

[Aho86] A.V. Aho, R. Sethi, J. D. Ullman "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986, 500 pp.

[Aho89] A.V. Aho, M. Ganapathi, S.W.K. Tjiang "Code Generation Using Tree Matching and Dynamic Programming", ACM Transactions on Programming Languages and Systems", Vol. 11, No. 4, 1989, pp. 491-516.

[Boehm88] H.-J. Boehm, M. Weiser "Garbage collection in an uncooperative environment", Software Practice & Experience, Vol. 18, No. 9, 1988, pp. 807-820.

[Boehm93] H.-J. Boehm "Space efficient conservative garbage collection", In Proceedings of the Conference on Programming Language Design and Implementation, 1993, pp. 197-206.

[Chanon75] R. Chanon "Compiler construction in an undergraduate course: some difficulties", ACM SIGCSE Bulletin, Vol. 7, No. 2, 1975, pp. 30-32.

[Crowe02] M.K. Crowe "Compiler Writing Tools Using C#", see http://cis.paisley.ac.uk/crow-ci0/

[Fraser92] C.W. Fraser, R.R. Henry, T.A. Proebsting "BURG – Fast Optimal Instruction Selection and Tree Parsing", SIGPLAN Notices, Vol. 27, No. 4, 1992, pp. 68-76.

[Gough02] J. Gough "Compiling for .NET Common Language Runtime", Addison-Wesley, 2002

[McConnell93] S. McConnell "Code Complete", Microsoft Press, 1993

[Muchnick97] S. Muchnick "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997, 856 pp.

[Pelegrì-Llopart88] E. Pelegrì-Llopart, S.L. Graham "Optimal Code Generation for Expression Trees: An Application of BURS Theory", Proceedings of the Conference on Principles of Programming Languages, 1988, pp. 294-308.

[Richter02] J. Richter "Applied .NET Framework Programming", Microsoft Press, 2002

[Stutz03] D. Stutz, T. Neward, G. Shilling "Shared Source CLI Essentials", O'Reilly, 2003