Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution

Vitaly V. Mikheev

Excelsior, LLC, Novosibirsk, Russia
vmikheev@excelsior-usa.com

Abstract. The XDS framework has been developed since 1993. During this time, it has been serving as a production compiler construction framework as well as a base for research projects. Completely written in Oberon-2, XDS uses a *mixed design* technique amalgamating modular and object-oriented approaches. It combines advantages of both approaches, at the same time avoiding drawbacks inherent to each of them if used separately. In this paper we set out how the mixed design approach contributes to extensibility of the framework with respect to including support for new input languages and target architectures and implementing new optimizations. In the first part of the paper we give an overview of the XDS framework architecture emphasizing which parts are worth applying the object-oriented design. In the second part, we describe our experience of extending XDS with support for the Java language and implementing interprocedural and object-oriented optimizations.

Keywords: object-oriented design, front-end, back-end, Java, performance, native code

1 Introduction

Most modern compilers have a multi-layer hierarchical architecture with many replaceable components. They also usually have many forms of intermediate representation (IR) of a program being compiled. However, the advantages that can be obtained from the system organization justify such a complexity. First, the architecture allows implematation of compiler for new input languages at a lower cost that is a good example of software reuse. In fact, a compiler designed in this way may be considered as a *compiler construction framework* rather than a solid application. This is the mainstream approach to compiler construction today and many production compilers have a similar architecture, for instance the IBM's XL compilers for FORTRAN, C/C++, and Java [5]. Another advantage of a multi-layer compiler design is the ability to produce code for different platforms (target architectures) that contributes to portable software development. The widespread virtual machine technology with portable byte-code is not the only way to achieve portability. It also may be supported in a multi-layer compiler construction framework by providing a component responsible for target-specific code generation for each platform[1] just like it is supported by providing a virtual machine for each of them. However, the multi-language and multi-platform support may lead to sacrificing performance of generated code that is unacceptable for production compilers. In the next sections we describe our solutions of that problem applied to the translation of Java to native compiled code. Finally, the multi-layer architecture allows adding new optimizations to the compiler. It is a very important point because code optimization is the subject of active investigations today. Using new optimizing techniques obtained from the latest research results is essential to construct "state-of-the-art" optimizing compilers.

The rest of the paper is organized as follows: Section 2 describes the XDS framework architecture, Section 3 focuses on the role of object-oriented design in compiler construction. Our experience of extending the XDS framework is described in Section 4. Finally, Section 5 summarizes the paper.

2 The XDS Framework Overview

XDS (eXtensible Development System) is a framework on which base several compilers have been developed. Among input programming languages supported by XDS are Modula-2, Oberon-2 [15], JVM bytecode, Java and two Nortel Networks' proprietary languages intended for telecommunication software development. Native code generation is implemented for the following target architectures: Intel x86, m68k, SPARC, PowerPC and VAX. As many other compilers, XDS also supports "via C" cross-compilation in order to be available for a wide variety of platforms. Furthermore, having the multi-component architecture, the XDS framework may be used for other purposes than compiler construction. The *XDS family tools* reusing particular framework components are built on top of XDS. A *converter from Modula-2 to C++* text has been developed for code migration purposes. The *static analyzer* [14] allows one to reveal run-time errors in large programs written in Modula-2, Oberon-2 or Java without execution. The *InterView source code browser*[18] is aimed at maintenance and reengineering of large-scale software systems. With its help, developers are able to find and visualize any program entity properties and non-trivial data-flow dependencies across the whole project. InterView supports a project database and a comprehensive query language. The database generation component is built into the XDS compilers for Modula-2, Oberon-2 and Java.

The XDS framework architecture is shown in Fig. 1. A common part of all tools built on XDS is the *project system* which is a supervisor responsible for system component management. In general, it takes a *system configuration file* and a *user project file* and invokes the respective framework components, for instance, compiling source files to object code. In particular, the project system is responsible for smart recompilation of out-of-date modules. After compilation, it produces a *response file* containing a list of files generated. The project

[1] Strictly speaking, a portable run-time support is also required, but this problem is much simpler than retargetable code generation so it is left out of the scope of this paper.

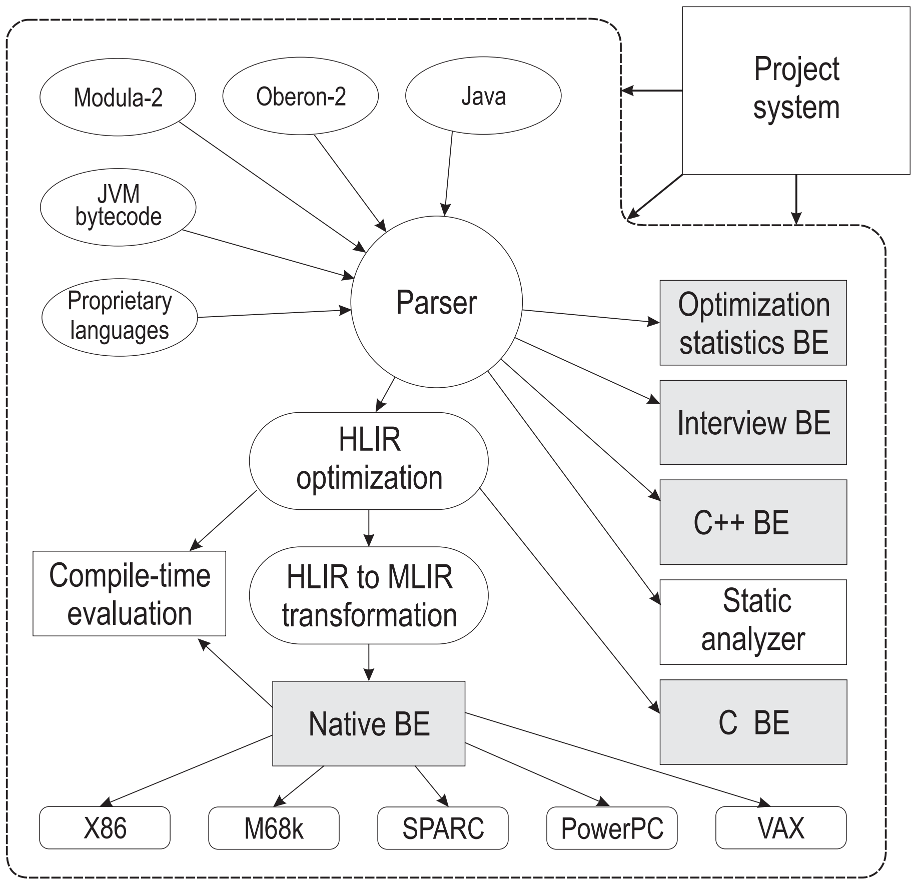

Fig. 1. XDS framework architecture

system may also use a template provided as part of system configuration to generate such a response file that allows XDS compilers to use any linker or maker to create executable file or perform other types of further processing, if required. Some points are worth noting here. The configuration of working system, including source languages, target platforms, code generation settings (e.g. optimization control, alignment etc.), is constructed on-the-fly. In other words, given configuration and project settings the project system composes the actual tool deciding on which components should be invoked. Note that such compiler organization allows generating object code for two "platforms" at the same time that makes sense, for instance, if the either platform is the InterView database.

In XDS, as in most modern compilers, there are separate front-end (FE) and back-end (BE) components. In general, FE parses source files to produce some kind of IR, and BE, in turn, takes the resulting IR and generates object code. However, strict meaning of the concepts usually varies from system to system. The following questions can help to make it clear:

1. What kind of intermediate representation does FE produce?
2. Is FE target independent?
3. Is BE language independent?

The answer to question 1 for XDS is the attributed syntax tree denoted in Fig. 1 as the high-level intermediate representation (HLIR). Other alternatives used in compiler construction research projects are either a stack-oriented bytecode of some virtual machine [10] or a low-level instruction representation [7]. Nowadays, it is widely acknowledged that the syntax tree keeps all information required for further optimizations which is usually lost in the other forms of IR [8, 9]. Different approaches have been proposed to solve the problem. One of them is to provide low-level representation with extra information, which can be used, then for some specific optimizations. In our opinion, this approach suffers from inability to incorporate new optimizations into the compiler without reimplementing the IR construction component. Another good reason to have IR as the syntax tree is that XDS back-ends are not limited to object code generation. For instance, the InterView data base generator and the static analyzer that we consider as XDS back-ends, demand IR to preserve as much information about source text as possible. In some cases, we had to carry out a large amount of work to meet that requirement. For example, our JVM bytecode FE uses a special technique based on symbolic computations to reconstruct the original syntax tree from JVM instructions and type information residing in JVM class files [1]. Some of related works [8, 11] do not follow this way and reconstruct syntax tree only partially. However, our experience shows that it results in less optimized code and decreases reusability of the FE component. Finally, the tree is the most appropriate IR for converter to C++. Thus, we strongly believe that the attributed syntax tree is the best choice for IR, which compiler's FE should produce.

The same reasoning may be taken into account when answering question 2. XDS FE is completely target-independent but in some of related works [10] FE performs target-depended jobs, e.g. field offset computation, constant evaluations depending on memory layout, etc. We deem the profits that can be obtained from such a decision do not outweigh the disadvantages that come with the lack of portability.

Another part of the XDS framework architecture is the HLIR optimizer. Its mission is to perform optimizations on the tree. Procedure inlining, constant propagation, object-oriented optimizations are implemented in the HLIR optimizer. After optimization, HLIR is converted to the middle-level intermediate representation (MLIR). This is implemented as an extra tree traversal and serves as a preliminary phase for native code generation. At this stage, the initial tree representation is simplified, for example, *synchronized block* in Java is transformed to *try-finally* block [2], thus reducing the number of different operators

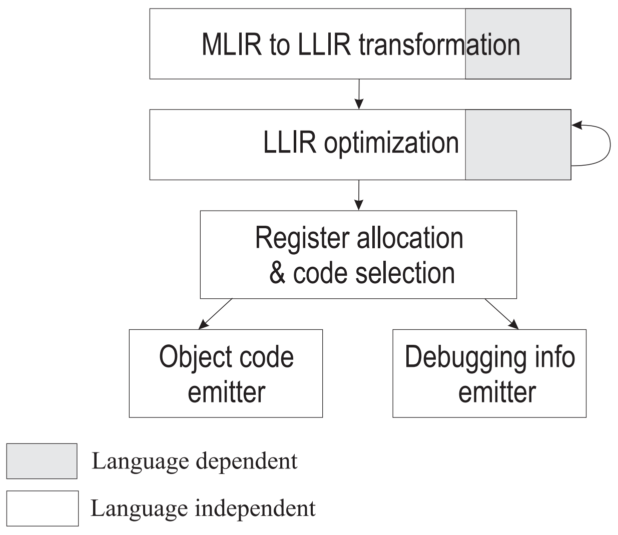

Fig. 2. XDS native BE organization

in MLIR. In particular, it allows BEs to be language-independent to the maximum extent because the conversion unifies operators from different languages to some common representation. MLIR, in turn, can be thought of as input language for native BE.

An ideal compiler architecture should have language-independent BE component. Theoretically, this goal may be achieved through appropriate BE design and it would be a good solution in case of supporting only one input language. Certainly, it is possible to implement a unified BE that takes MLIR generated by any language-specific FE and produces code but at cost of worse code performance. Several significant examples are given in Section 4. Thus, question 3 has the following answer: XDS native BEs contain language-dependent parts.

Fig. 2 shows a common structure of XDS native BE. At first stage, MLIR is converted to the triad form [12] called the low-level intermediate representation (LLIR). In fact, LLIR can be considered as an abstract (target-independent) low-level instruction set. First, BE performs some simple optimizations just on the representation, after that it is transformed to SSA (Static Single Assignment) form to make the majority of optimizations such as common subexpression elimination, dead code removal, etc. At the next phase LLIR is converted into the DAG (Direct Acyclic Graph) form (set of trees) to accomplish an optimal code selection with the bottom up tree rewriting technique [6]. We use the BURG tool that produces rewriters in Oberon-2 from description of the rewriting grammars representing the target architecture (instruction set and addressing modes). Each grammar rule is attributed with a "rule cost" that depends on target architecture as well, so a rewriter always tries to minimize the entire inference cost. Along with DAG rewriting, graph coloring-based register allocation is performed. Finally, code and debugging information emitters are invoked to produce object code or assembler source text.

3 Object-Oriented Design in Compiler Construction

At present, object-oriented design (OOD) is widely used in the software industry. Adherents of OOD assert that it offers better solutions for typical problems arising during software life cycle. In particular, OOD eases program code maintainability and extensibility, increasing the degree of software reuse. Taking the above reasons into account we, however, do not share the opinion that OOD is the panacea for large system design problems. Many requirements that such systems have to meet may be fulfilled with help of the traditional modular design approach. In our opinion, the bias to object-oriented programming and popularity of C++ are to some extent caused by the availability of some modular means in C++ which are not present in pure C. In fact, many C programmers have missed the modular stage of their programming methodology evolution and found themselves right in the object-oriented stage. Really, one should be a strong-will person to write in C according to modular programming principles.

Our design approach is not a conventional OOD, although some of related works recognize it as a preferable way in compiler construction [4]. We strongly believe it would be wrong to use OOD for the whole system being designed without special consideration of which components are worth applying OOD and which are not. Thus, we keep the golden mean in design issues: on the one hand, we remain staunch supporters of the modular design approach which is a good way to achieve the main goal of large system design: *to be able to manage complexity*; on the other hand, we accept the OO approach as a very useful mean to improve reusability and extensibility of particular system components. An outline of the approach that we call *mixed design* looks as follows:

1. Design starts as the traditional modular approach: one defines module data encapsulation and import-export relationships
2. If behavior of some previously designed module should be corrected at runtime to fulfil system requirements, but module interface remains unchangeable, then the module is converted to a class straightforward: module global variables become instance fields and procedures make methods.
3. As a rule, aggregate data types (RECORD types in terms of Oberon-2) are used to represent entities from problem domain. If such a type requires polymorphic processing, it is immediately converted to class. Note that it is usually quite obvious whether a data type is worth using polymorphism: for the lack of OO features polymorphism is usually simulated with a tag field that is checked in implementations of data type operations. The rule is to avoid such implementation technique and use OO mechanisms instead. The same reasoning concerns extending a RECORD type with new fields: in languages without OO support, it is implemented by casting pointers to different records from within data type operations.

Thus, according to rule 3, we use OO features just to implement data types in a natural way. In other words, *we distinguish the module notion from the data type notion* whereas in OOD they are both merged into the single class notion. We think this design technique may be successfully applied to a broad variety of projects and our own experience shows that it is sufficient to design a compiler construction framework meeting the language and target independence requirement. We suppose this is quite a representative large project. Several significant examples of object-oriented compiler construction are given here.

– The abstract syntax tree has to be designed in OO manner because there is a lot of polymorphic processing involving it: tree traversals for different purposes, control graph markup and analysis, polymorphic containers with HLIR objects for static analysis, IR conversions and so on. Besides, each tree object has specific instance fields for its own that can be implemented by inheritance in a natural way.
– FE and BE components configured on-the-fly in a multi-language retargetable compiler are worth implementing as instance objects of the FE and BE classes, according to rule 2.
– As shown in Fig. 1, the compile-time evaluation block is designed as a distinct component providing its service for both FE and BE. FE usually performs constant propagation on the syntax tree whereas BE requires the capability of constant evaluation when optimizing the SSA form of LLIR. Note that rules of compile-time evaluation vary for different languages. For instance, in Modula- 2/Oberon-2, once integer overflow has happened during constant evaluation, it has to be considered as a compile-time error whereas in Java, it just should be computed as a 2's complement even without issuing a warning. There are many other differences so the constant evaluation module is a good candidate for conversion to class by rule 2.
– Some other components of the XDS frameworks, for example object code and debugging information emitters (see Fig. 1), are also designed as instance objects of some class depending on the system configuration.

Summarizing this section we would like to emphasize that we have profited from the *motivated* usage of OO mechanisms in compiler design and development process.

4 The XDS Evolution

The primary goal of the XDS framework implementation was to bring high optimizing compiler solutions to Modula-2 and Oberon-2 (M2/O2) programmers. Nonetheless, from the very beginning XDS was designed with respect to software reuse so it was not limited to compilation of specific languages. Finished working at the M2/O2 compilers, we were challenged to apply the XDS technology to another language and Java was chosen as the next step. Initially, we supposed that the implementation of a Java compiler requires the development of the respective FE and introducing some minor modifications into native BE.

However, we *considerably* underestimated the amount of work involved: many BE optimization algorithms proved to be language-dependent with respect to either correct code generation or performance. In fact, at the sacrifice of performance, we would be able to implement a Java to native code compiler with less efforts, but the usefulness of such a compiler would be doubtful. In this section we describe our experience of extending the XDS framework with the support for Java and implementation of new optimization techniques which are now available in all compilers built on the XDS framework.

4.1 Java Implementation Notes

Despite many similarities between Oberon-2 and Java, implementation of many language features had required significant efforts. It is debatable if they may be considered as advantages of the Oberon-2 facilities or not, but anyway we had to implement them carefully to meet the Java Language Specification.

Cyclic Import Cyclic import is prohibited in Oberon-2, but not in Java, and it spoiled the separate compilation technique used in XDS compilers before. Apart from smart recompilation of the whole project, every single module was compiled by the simple two-step scheme: firstly, module is parsed to build the syntax tree and then BE is invoked to produce object code. In order to support separate compilation, XDS compiler creates a precompiled version of each module, so-called *symbol file*. All required information on the entities exported from the module is resided in its symbol file: exported types, procedures, variables, etc. Thus, if a module A, being compiled, imports a module B, the compiler just reads *B.sym* file to complete the syntax tree for A with the entities from B. The acyclic import graph always guarantees that each module may be compiled separately. In Java, the same technique may be applied only if module does not use cyclic import, otherwise all the modules which constitute such a cycle of the import graph have to be compiled at the same time. Fortunately, the XDS architecture did not require significant reengineering, because the modified compilation scheme was easily implemented in the project system — the supervisor responsible for invocation of components as described in Section 3.

Address Arithmetic Although Modula-2 does not encourage address arithmetic and unrestricted type casting, it still leaves programmer a loophole to write code in such a way. Just opposite to Modula-2, Java imposes "pure type" programming with no low-level operations. In addition, Java does not allow reference variables to point to values of primitive types. Those restrictions may have strong influence on the performance of generating code, especially with respect to pointer aliasing and register allocation algorithms. Thus, implementation of a common Java/Modula-2 BE would result in poorer Java code performance. We have modified the optimization algorithms that can benefit from the absence of address arithmetic.

Bytecode Input Files It is known that Java applications and class libraries are distributed in the form of *.class* (JVM bytecode) files. This new deployment technology leaves the door open for further optimization of both Java applications and third-party libraries they use. It is a unique opportunity to achieve high degree of optimization because programming languages and deployment technologies used before did not allow libraries to be optimized. Really, C/C++ or Fortran libraries are usually distributed in some form of object code so compilers can not perform global analysis and optimization of entire application including libraries. In order to use that opportunity, Java compilers should be able to compile *.class* files just like they compile other source files. For that purpose, the JVM bytecode front-end component of the XDS framework has been developed. In addition, the XDS project system was extended in order to be able to process bytecode files.

Exception Handling There are good reasons to implement exception handling for Java in a different way. First of all, according to the ISO Modula-2 standard [3], try-block must (syntactically) coincide with procedure block whereas in Java, it may comprehend an arbitrary set of operators [2]. It has a strong influence on optimization at presence of exceptions, especially with respect to register allocation. The matter is that Java exception handling makes program control graph complicated and requires a redesign of the BE algorithm determining life time of local variables. Otherwise it may result in poor performance or even incorrect generated code. Another point is intensive exploitation of exception handling in fine-grain Java methods. The simple *setjmp/longjmp* technique implemented for Modula-2 has some performance overhead for each try-block and being applied to Java makes performance twice as bad. Instead, we have implemented the other, so-called *frame-to-frame propagation* technique [16] with some modifications that results in no try-block overhead at all in the case of non-exceptional execution. The last solves the performance problem, but the cost is code size increasing, which may be unacceptable for Modula-2 — the language often used in software development for embedded systems.

Floating-Point Arithmetic According to the IEEE 754/854 standard [17] to which Java complies, no floating-point (FP) operations cause an exception. For instance, 1/0 or even 0/0 are well-defined FP values. It may seem to be curious but they are even typed values, for instance there exists 0/0 of the *double* FP type. The matter is that IEEE 754/854 introduces two new binary-coded values: signed infinity and NaN (not a number). Both of them are supported in Java, so you may find a piece of Java code like this:

```
static final float negInfinity = -1.0/0;
```

As a consequence, some ordinary symbolic computations become incorrect, for instance $0 \cdot x = 0$ because $0 \cdot \infty = NaN$. The required modifications have been made in the constant evaluation component which is language-specific as described in Section 3. But the real Java challenge for the XDS BE optimizer was that the axiom

$$x \leq y \Leftrightarrow \neg(x > y)$$

is no longer valid. No, this is not the three-value logic, just result of comparison between any FP value and NaN is always false. We have modified the optimizer to make it IEEE 754/854 compliant. Thus, it is quite obvious that FP optimizations may not be implemented for Oberon-2 and Java in the same way. The last curious thing is a very suspicious IEEE 754 feature, signed zero(!). In fact, there exist both $+0$ and -0 as distinct binary-coded values such that $+0 = -0$. Only during Java implementation we finally understood what is the difference between them:

$$1/+0 = +\infty$$

$$1/-0 = -\infty$$

Heap Objects A known disadvantage of Java applications is exhaustive dynamic memory consumption. For the lack of stack objects — class instances put on the stack frame, all objects have to be allocated on the heap by the *new* operator. Presence of Java class libraries makes the situation much worse because any service provided by some library class requires the allocation of its instance object. To overcome the drawback we have implemented a static analysis that allows the compiler to define the life time of dynamically allocated objects and allocate short living objects on the stack. Note that the optimization technique saves both memory and time resources, because stack allocation actually happens at compile-time rather than at run-time and the garbage collector does not have to care about such objects after they die.

Multi-Threading Java has built-in multi-threading support that allows instance methods to be synchronized if they operate on the same object simultaneously from different threads. Many library classes are designed with respect to a possible use in a multi-threaded environment, so even fine-grain methods have the *synchronized* specifier [2]. As a consequence, straightforward implementation of synchronization support results in a dramatic performance degradation. In order to ensure an acceptable performance level, we have designed a special technique, so-called *lightweight synchronization*, that engages the actual synchronization only if several threads actually want to operate on the same object. Note that the implementation of lightweight synchronization is only possible with support from native BE.

The main conclusion we have drawn after development of the Java to native code compiler is that *in order to achieve the high degree of code optimization, some parts of native BE have to be language-specific.*

4.2 Optimizations

It is widely acknowledged that code written in OO programming languages has some performance overhead due to dynamic method dispatch inherent to the languages. Being implemented by VMT (Virtual Method Table), virtual method invocation does not take much processor time itself but it hinders compiler to use the traditional technique of interprocedural optimizations, such as inlining. Type inference [13] is a static type analysis that allows compiler to replace virtual calls with direct ones safely and, thus, to improve performance considerably. The characteristic feature of type inference is so-called *polyvariant* analysis that typically performs repeated traversals of method control graph from within different calling contexts. Note that even with virtual calls replaced we still need to have the syntax trees of all methods at our disposal, for instance, to perform intermodule procedure inlining or analyze the life time of dynamically allocated objects as described above. For that purpose, we have implemented *syntax tree object persistency* allowing an arbitrary IR object graph to be saved to/restored from file. This technique resembles the slim binaries approach [9] used for dynamic compilation although we restrict its use to static code analysis and optimization only. Since real world Java applications use OO features extensively, the implementation of OO optimizations allow us to improve performance of Java code to a great extent. Benchmark results and other information related to the XDS Native Java project may be found at [19].

5 Conclusion

This paper has presented a technique of multi-language, multi-platform compiler construction. The technique has been used to integrate quite different programming languages such as Modula-2, Oberon-2 and Java into a single compiler construction framework. Our results refute the opinion that compiler should be specially designed for certain programming language in order to produce high performance code. The work has shown that language and platform independence and high performance do not contradict, although they require specific architectural decisions. The interesting direction for future works is to investigate a technique of seamless integration between Java and other languages, such as Modula-2, and also probably C. As a rule, some part of a typical Java application called *native methods* is written in other languages so Java programmers are compelled to use several tools including compilers in the development process. It would be interesting to design and implement an integrated compiler environment having all required means of Java application development.

References

1. T. Lindholm, F. Yellin, B. Joy, K. Walrath: *The Java Virtual Machine Specification*. Addison-Wesley, 1996
2. J. Gosling, B. Joy and G.Steele: *The Java Language Specification*. Addison-Wesley, Reading, 1996
3. *The Modula-2 language international standard*. ISO/EEC 10514, 1994
4. J. Holmes: *Object-Oriented Compiler Construction*. Prentice- Hall, 1995
5. *IBM Visual Age family products*. **http://www-4.ibm.com/software/ad/**

6. K J. Gough: *Bottom-up tree rewriting tool MBURG.* SIGPLAN Notices, Volume 31, Number 1, 1996
7. J. Dean, G. DeFouw et al: *Vortex: An Optimizing Compiler for Object-Oriented Languages.* In Proc. of OOPSLA'96, San Jose, CA, October, 1996
8. H. Saito, N. Stravrakos et al: *The Design of the PROMIS Compiler*, In Proc. 8th International Conference, Compiler Construction. Volume 1575 of LNCS, Springer-Verlag, 1999
9. M. Franz, Th. Kistler: *Slim binaries.* Technical report 96-24, Department of Information and Computer Science, UC Irvine, 1996
10. K J. Gough: *Multi-language, Multi-target Compiler Development: Evolution of the Gardens Point Compiler Project.* In Proc. Joint Modular Languages Conference, JMLC'97. Volume 1204 of LNCS, Springer-Verlag, 1997
11. H. Kienle: *j2s: A SUIF Java Compiler.* Technical Report TRCS98-18, Computer Science Department, University of California, Santa Barbara, 1998
12. A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986
13. O. Agesen: *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism.* In Proc. ECOOP'95, Aarhus, Denmark, 1995
14. V. Shelekhov, S. Kuksenko: *On the Practical Static Checker of Semantic Run-time Errors.* In Proc. of the 6th Asia Pacific Software Engineering Conference APSEC'99, Japan. IEEE Computer Society Press, 1999
15. N. Wirth, M. Reiser: *Programming in Oberon. Steps beyond Pascal and Modula.* Addison-Wesley, 1992
16. J. Lajoie: *Exception Handling — supporting the runtime mechanism,* C++ Report, Vol.6, No. 3, 1994.
17. *Pentium Pro Family Developer's Manual. Volume 2: Programmer's Reference Manual, Chapter 7, Floating-Point Unit*, Intel Corporation, 1996
18. *The InterView Source Code Browser.* User's Guide.
http://www.excelsior-usa.com/intervew.html
19. *D. Leskov. XDS Native Java.* Whitepaper.
http://www.excelsior-usa.com/jetwp.html