Intermediate Representation

view issue

by Fred Chow | November 22, 2013

Topic: Programming Languages

Tweet 17

Like 13 people like this. Sign Up to see wh

The Challenge of Cross-language Interoperability

The increasing significance of intermediate representations in compilers

FRED CHOW

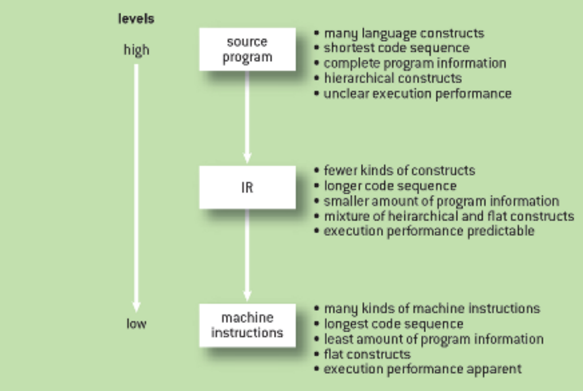
Program compilation is a complicated process. A compiler is a software program that translates a high-level source language program into a form ready to execute on a computer. Early in the evolution of compilers, designers introduced IRs (intermediate representations, also commonly called intermediate languages) to manage the complexity of the compilation process. The use of an IR as the compiler's internal representation of the program enables the compiler to be broken up into multiple phases and components, thus benefiting from modularity.

An IR is any data structure that can represent the program without loss of information so that its execution can be conducted accurately. It serves as the common interface among the compiler components. Since its use is internal to a compiler, each compiler is free to define the form and details of its IR, and its specification needs to be known only to the compiler writers. Its existence can be transient during the compilation process, or it can be output and handled as text or binary files.

THE IMPORTANCE OF IRS TO COMPILERS

An IR should be general so that it is capable of representing programs translated from multiple languages. Compiler writers traditionally refer to the semantic content of programming languages as being high. The semantic content of machine-executable code is considered low because it has retained only enough information from the original program to allow its correct execution. It would be difficult (if not impossible) to re-create the source program from its lower form. The compilation process entails the gradual lowering of the program representation from high-level human programming constructs to low-level real or virtual machine instructions (figure 1). In order for an IR to be capable of representing multiple languages, it needs to be closer to the machine level to represent the execution behavior of all the languages. Machine-executable code is usually longer because it reflects the details of the machines on which execution takes place.

FIGURE 1 The Different Levels of Program Representations

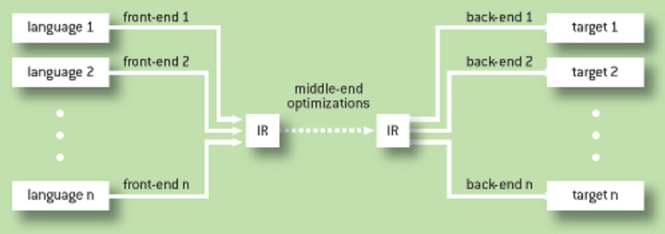

levels
high
low
source program
IR
machine instructions
• many language constructs
• shortest code sequence
• complete program information
• hierarchical constructs
• unclear execution performance
• fewer kinds of constructs
• longer code sequence
• smaller amount of program information
• mixture of heirarchical and flat constructs
• execution performance predictable
• many kinds of machine instructions
• longest code sequence
• least amount of program information
• flat constructs
• execution performance apparent

A well-designed IR should be translatable into different forms for execution on multiple platforms. For execution on a target processor or CPU, it needs to be translated into the assembly language of that processor, which usually is a one-to-one mapping to the processor's machine instructions. Since there are different processors with different ISAs (instruction set architectures), the IR needs to be at a higher level than typical machine instructions, and not assume any special machine characteristic.

Using an IR enables a compiler to support multiple front ends that translate from different programming languages and multiple back ends to generate code for different processor targets (figure 2). The execution platform can also be interpretive in the sense that its execution is conducted by a software program or virtual machine. In such cases, the medium of execution can be at a level higher than assembly code, while being lower or at the same level as the IR.

FIGURE 2

A Compiler System Supporting Multiple Languages and Multiple Targets



The adoption of IRs enables the modularization of the compilation process into the front end, the middle end, and the back end. The front end specializes in handling the programming language aspects of the compiler. A programming language implementer only needs to realize the accurate translation of the language to an IR before declaring the work complete.

The back end takes into account the particulars of the target machine and translates the IR into the machine instructions to be executed on the hardware. It also transforms the code to take advantage of any hardware features that benefit performance. By starting its translation from the IR, the back end in effect supports the different languages that produce the IR.

The middle end is where target-independent optimizations take place. The middle-end phases perform different transformations on the IR so the program can run more efficiently. Because optimizations on the IR usually benefit all targets, the IR has significant role of performing target-independent optimizing transformations. In this role, its design and specification become even more important. It needs to encode any source program information that is helpful to the optimization tasks. The IR's design has a bearing on optimization efficiency, and optimization is the most time-consuming part of the compilation process. In modern-day compilers, the IR dictates the infrastructure and overall engineering of the compiler. Any major change to the IR could imply a substantial overhaul in the compiler implementation.

THE DIFFERENT IR FORMS

The minimal requirement of an IR is to provide enough information for the correct execution of the original program. Each instruction in an IR typically represents one simple operation. An IR should have fewer kinds of constructs than any typical programming language, because it does not need to be feature-rich to facilitate programming use by humans. Compilers like to see the same programming constructs or idioms being translated to uniform code sequences in the IR, regardless of their source languages, programming styles, or the ways the programmers choose to code them. Imposing canonical forms in IRs reduces the variety of code patterns that the compiler has to deal with in performing code generation and optimization. Because of its finer-grained representation of the program, an IR's instructions may map many-to-one to a machine instruction because one machine instruction may perform multiple operations, as in multiply-add or indexed addressing.

The form of an IR can be classified as either hierarchical or flat. A hierarchical IR allows nested structures. In a typical programming language, both program control flows (e.g., if-then-else, do-loops) and arithmetic expressions are nested structures. A hierarchical IR is thus closer in form to the typical programming language, and is regarded as being at a higher level. A hierarchical IR can be represented internally in the form of trees (the data structure preferred by compilers) without loss of accuracy.

A flat IR is often viewed as the instructions of an abstract or virtual machine. The instructions are executed sequentially, as in a typical processor, and control flows are specified by branch or jump instructions. Each instruction takes a number of operands and produces a result. Such IRs are often specified as compilation targets in the teaching of compiler construction.

Lying somewhere between hierarchical and flat IRs is the language of an abstract stack machine. In a stack machine, each operand for arithmetic computation is specified by an instruction that pushes the operand onto the stack. Each arithmetic expression evaluation is done on the operands that are popped off the top of the stack, and the subsequent result is pushed back onto the stack. The form of the IR is flat, with control flow represented by explicit branch instructions, but the instruction sequence for arithmetic computation can be regarded as corresponding to the reverse Polish notation, which can be easily represented internally in a tree data structure. Using the language of a stack machine as the IR has been a common practice from the first IR defined for Pascal, called p-code,[1] to the current-day Java bytecode[6] or CIL (Common Intermediate Language[2]).

There is information complementary to the IR that serves purposes other than representing code execution. The compiler compiles the namespace in the original program into a collection of symbol names. Variables, functions, and type information belong to these symbol tables, and they can encode information that governs the legality of certain optimizing transformations. They also provide information needed by various tools such as debuggers and program analyzers. The symbol tables can be considered adjuncts to the IRs.

C has been used as the translation target of many programming languages because of its widespread use as a system programming language and its ability to represent any machine operation. C can be regarded as an IR because of its lower level relative to most languages, but it was not designed for easy manipulation by compilers or to be directly interpreted. In spite of this, many IRs have been designed by closely modeling the C language semantics. In fact, a good IR can be constructed by carefully stripping away C's high-level control-flow constructs and structured data types, leaving behind only its primitives. Many IRs can also be translated to C-like output for easy perusal by compiler developers. Such C-like IRs, however, usually cannot be translated to C programs that can be recompiled because of C's deficiencies in representing certain programming concepts such as exception handling, overflow checking, or multiple entry points to a function.

IRS FOR PROGRAM DELIVERY

With the widespread use of networked computers, people soon understood the advantage of an execution medium that is processor- and operating-system-neutral. The distribution and delivery process is easier with programs that can run on any machine. This write-once, run-anywhere approach can be realized with the virtual machine execution model to accommodate the diversity of system hardware.

Interpretive execution contributes to some loss of performance compared with compiled execution, and initially it made sense only for applications that are not computation-intensive. As machines become faster and faster, however, the advantages of the write-once, run-anywhere approach outweigh potential performance loss in many applications. This gave rise to the popularity of languages such as Java that can be universally deployed. The Java language defines the Java bytecode, which is a form of IR, as its distribution medium. Java bytecode can be run on any platform as long as the JVM (Java virtual machine) software is installed. Another example is CIL, which is the IR of the CLI (Common Language Infrastructure) runtime environment used by the .NET Framework.

With the growth of the mobile Internet, applications are often downloaded to handheld devices to be run instantly. Since IRs take up less storage than machine executables, they reduce network transmission overhead, as well as enabling hardware-independent program distribution.

JUST-IN-TIME COMPILATION

As the virtual machine execution model gained widespread acceptance, it became important to find ways of speeding up the execution. One method is JIT (just-in-time) compilation, also known as dynamic compilation, which improves the performance of interpreted programs by compiling them during execution into native code to speed up execution on the underlying machine. Since compilation at runtime incurs overhead that slows down the program execution, it would be prudent to take the JIT route only if there is a high likelihood that the resultant reduction in execution time more than offsets the additional compilation time. In addition, the dynamic compiler cannot spend too much time optimizing the code, as optimization incurs much greater overhead than translation to native code. To restrain the overhead caused by dynamic compilation, most JIT compilers compile only the code paths that are most frequently taken during execution.

Dynamic compilation does have a few advantages over static compilation. First, dynamic compilation can use realtime profiling data to optimize the generated code more effectively. Second, if the program behavior changes during execution, the dynamic compiler can recompile to adjust the code to the new profile. Finally, with the prevalent use of shared (or dynamic) libraries, dynamic compilation has become the only safe means of performing whole program analysis and optimization, in which the scope of compilation spans both user and library code. JIT compilation has become an indispensable component of the execution engines of many virtual machines that take IRs as input. The goal is to make the performance of programs built for machine-independent distribution approach that of native code generated by static compilers.

In recent years, computer manufacturers have come to the realization that further increases in computing performance can no longer rely on increases in clock frequency. This has given rise to special-purpose processors and coprocessors, which can be DSPs (digital signal processors), GPUs, or accelerators implemented in ASICs (application-specific integrated circuits) or FPGAs (field-programmable gate arrays). The computing platform can even be heterogeneous, where different types of computation are handed off to different types of processors, each having different instruction sets. Special languages or language extensions such as CUDA,[3] OpenCL,[8] and HMPP (Hybrid Multicore Parallel Programming),[4] with their underlying compilers, have been designed to make it easier for programmers to derive maximum performance in a heterogeneous setting.

Because these special processors are designed to increase performance, programs must be compiled to execute in their native instructions. As the proliferation of special-purpose hardware gathered speed, it became impossible for a compiler supplier to provide customized support for the variety of processors that exist in the market or are about to emerge. In this setting, the custom hardware manufacturer is responsible for providing the back-end compiler that compiles the IR to the custom machine instructions, and platform-independent program delivery has become all the more important. In practice, the IR can be compiled earlier, at installation time or at program loading, instead of during execution. Nowadays, the term *AOT* (ahead-of-time), in contrast with JIT, characterizes the compilation of IRs into machine code before its execution. Whether it's JIT or AOT, however, IRs obviously play an enabling role in this new approach to providing high-performance computing platforms.

STANDARDIZING IRS

So far, IRs have been linked to individual compiler implementations because most compilers are distinguished by the IRs they use. IRs are translatable, however, and it is possible to translate the IR of compiler A to that of compiler B, so compiler B can benefit from the work in compiler A. With the trend toward open source software in the past two decades, more and more compilers have been open sourced.[9] When a compiler becomes open source, it exposes its IR definition to the world. As the compiler's developer community grows, it has the effect of promoting its IR. Using an IR, however, is subject to the terms of its compiler's open source license, which often prohibits mixing it with other types of open source licenses. In case of licensing conflicts, special agreements need to be worked out with the license providers before such IR translations can be realized. When realized, IR translation enables collaboration between compilers.

Java bytecode is the first example of an IR with an open standard definition that is independent of compilers, because JVM is so widely accepted that it has spawned numerous compiler and VM implementations. The prevalence of JVM has led to many other languages being translated to Java bytecode,[7] but because it was originally defined to serve only the Java language, support for high-level abstractions not present in Java is either not straightforward or absent. This lack of generality limits the use of Java bytecode as a universal IR.

Because IRs can solve the object-code compatibility issue among different processors by simplifying program delivery while enabling maximum compiled-code performance on each processor, standardizing on an IR would serve the computing industry well. Experience tells us that it takes time for all involved parties to agree on a standard; most existing standards have taken years to develop, and sometimes, competing standards take time to consolidate into one. The time is ripe to start developing an IR standard. Once such a standard is in place, it will not stifle innovation as long as it is being continuously extended to capture the latest technological trends.

A standard IR will solve two different issues that have persisted in the computing industry:

• **Software compatibility.** Two pieces of software are not compatible when they are in different native code of different ISAs. Even if their ISAs are the same, they can still be incompatible if they have been built using different ABIs (application binary interfaces) or under different operating systems with different object file formats. As a result, many different incompatible software ecosystems exist today. The computing industry would be well served by defining a standard software distribution medium that is acceptable by most if not all computing platforms. Such a distribution medium can be based on the IR of an abstract machine. It will be rendered executable on a particular platform through AOT or JIT compilation. A set of compliance tests can be specified. Software vendors will need to distribute their software products only in this medium. Computing devices supporting this standard will be able to run all software distributed in this form. This standardized software ecosystem will create a level playing field for manufacturers of different types of processors, thus encouraging innovation in hardware.

• **Compiler interoperability.** The field of compilation with optimization is a conundrum. No single compiler can claim to excel in everything. The algorithm that a compiler uses may work well for one program but not so well for another. Thus, developing a compiler requires a huge effort. Even for a finished compiler, there may still be endless enhancements deemed desirable. Until now, each production-quality compiler has been operating on its own. This article has discussed IR translation as a way of allowing compilers to work together. A standard IR, if adopted by compiler creators, would make it possible to combine the strengths of the different compilers that use it. These compilers will no longer need to incorporate the full compilation functionalities. They can be developed and deployed as compilation modules, and their creators can choose to make the modules either proprietary or open source. If a compiler module wants to use its own unique internal program representation, it can choose to use the standard IR only as an interchange format. A standard IR would lower the entry barrier for compiler writers, because their projects could be conceived at smaller scales, allowing each compiler writer to focus on his or her specialties. An IR standard would also make it easier to do comparisons among the compilers because they would produce the same IR as output, which will lead to more fine-tuning. An IR standard could revolutionize today's compiler industry and would serve the interests of compiler writers very well.

Two visions for an IR standard are outlined here: the first is centered on the computing industry, the second on the compiler industry. The first emphasizes the virtual machine aspect, and the second focuses on providing good support to the different aspects of compilation. Because execution requires less program information than compilation, the second goal will require greater content in the IR definition compared with the first goal. In other words, an IR standard that addresses the first goal may not fulfill the needs of the second. It is also hard to say at this point whether one well-defined IR standard can fulfill both purposes at the same time.

The HSA (Heterogeneous System Architecture) Foundation was formed in 2012 with the charter of making programming heterogeneous devices dramatically easier by putting forth royalty-free specifications and open source software.[5] Its members intend to build a heterogeneous software ecosystem rooted in open royalty-free industry standards.

Recently, the foundation put forth a specification for HSAIL (HSA Intermediate Language), which is positioned as the ISA of an HSAIL virtual machine for any computing device that plans to adhere to the standard. HSAIL is quite low level, somewhat analogous to the assembly language of a RISC machine. It assumes a specific program and memory model catering to heterogeneous platforms where multiple ISAs exist, with one specified as the host. It also specifies a model of parallel processing as part of the virtual machine.

Although HSAIL is aligned with the vision of enabling a software ecosystem based on a virtual machine, its requirements are too strong and lack generality, and thus will limit its applicability to the specific segment of the computing industry that it targets. Though HSAIL is meant as the compilation target for compiler developers, it is unlikely that any compiler will adopt HSAIL as an IR during compilation because of the lack of simplicity in the HSAIL virtual machine. It is a step in the right direction, however.

IR DESIGN ATTRIBUTES

In conclusion, here is a summary of the important design attributes of IRs and how they pertain to the two visions discussed here. The first five attributes are shared by both visions.

• **Completeness.** The IR must provide clean representation of all programming language constructs, concepts, and abstractions for accurate execution on computing devices. A good test of this attribute is whether it is easily translatable both to and from popular IRs in use today for various programming languages.

• **Semantic gap.** The semantic gap between the source languages and the IR must be large enough that it is not possible to recover the original source program, in order to protect intellectual property rights. This implies the level of the IR must be low.

• **Hardware neutrality.** The IR must not have built-in assumptions of any special hardware characteristic. Any execution model apparent in the IR should be a reflection of the programming language and not the hardware platform. This will ensure it can be compiled to the widest range of machines, and implies that the level of the IR cannot be too low.

• **Manually programmable.** Programming in IRs is similar to assembly programming. This gives programmers the choice to hand-optimize their code. It is also a convenient feature that helps compiler writers during compiler development. A higher-level IR is usually easier to program.

• **Extensibility.** As programming languages continue to evolve, there will be demands to support new programming paradigms. The IR definition should provide room for extensions without breaking compatibility with earlier versions.

From the compiler's perspective, there are three more attributes that are important considerations for the IR to be used as a program representation during compilation:

• **Simplicity.** The IR should have as few constructs as possible while remaining capable of representing all computations translated from programming languages. Compilers often perform a process called canonicalization that massages the input program into canonical forms before performing various optimizations. Having the fewest possible ways of representing a computation is actually good for the compiler, because there are fewer code variations for the compiler to cover.

• **Program information.** The most complete program information exists in the source form in which the program was originally written, some of which is derived from programming language rules. Translation out of the programming language will contribute to information loss, unless the IR provides ways of encoding the escaped information. Examples are high-level types and pointer aliasing information, which are not needed for program execution but affect whether certain transformations can be safely performed during optimization. A good IR should preserve any information in the source program that is helpful to compiler optimization.

• **Analysis information.** Apart from information readily available at the program level, program transformations and optimizations rely on additional information generated by the compiler's analysis of the program. Examples are data dependency, use-def, and alias analysis information. Encoding such information in the IR makes it usable by other compiler components, but such information can be invalidated by program transformations. If the IR encodes such analysis information, it needs to be maintained throughout the compilation, which puts additional burdens on the transformation phases. Thus, whether or not to encode information that can be gathered via program analysis is a judgment call. For the sake of simplicity, it can be left out or made optional.

A standard for a universal IR that enables target-independent program binary distribution and is usable internally by all compilers may sound idealistic, but it is a good cause that holds promise for the entire computing industry.

REFERENCES

1. Barron, D. W. (Ed.). 1981. *Pascal–The Language and its Implementation.* John Wiley.

2. CIL (Common Intermediate Language); http://en.wikipedia.org/wiki/Common_Intermediate_Language.

3. CUDA; http://www.nvidia.com/object/cuda_home_new.html.

4. HMPP; http://www.caps-entreprise.com/openhmpp-directives/.

5. HSA Foundation; http://www.hsafoundation.com/.

6. Java bytecode; http://www.javaworld.com/jw-09-1996/jw-09-bytecodes.html.

7. JVM languages; http://en.wikipedia.org/wiki/List_of_JVM_languages.

8. OpenCL; http://www.khronos.org/opencl/.

9. Open source compilers; http://en.wikipedia.org/wiki/List_of_compilers#Open_source_compilers.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

Fred Chow (chowfred@icubecorp.com) pioneered the first optimizing compiler for RISC processors, the MIPS Ucode compiler. He was the chief architect behind the Pro64 compiler at SGI, later open sourced as the Open64 compiler. He later created the widely accepted PathScale version of the Open64 compiler. Algorithms he developed have been widely adopted in today's compilers. He is currently leading the compiler effort for a new processor at ICube Corp. He received a B.S. degree from the University of Toronto and M.S. and Ph.D. degrees from Stanford University.

Originally published in Queue vol. 11, no. 10—
see this item in the ACM Digital Library

COMMENTS

raiph | Mon, 02 Dec 2013 23:53:48 UTC

```
Here's how NQP's QAST (https://github.com/perl6/nqp/blob/master/docs/qast.markdown) informally stacks up against your bullet
points:

" Completeness. Written in NQP. NQP is a simple functional/OO lang for writing compilers. A QAST tree consists of a tree of
NQP objects made from about a dozen classes corresponding to very generic intermediate things (eg compilation units,
closures, calls of closures, literal values, ops).

" Semantic gap. Er, no idea.

" Hardware neutrality. NQP is designed to be highly portable. It targets both existing VMs (JVM, CLR, etc.) and its own
custom VM (MoarVM) which is itself highly portable.

" Manually programmable. QAST is just a tree of object declarations written in NQP.

" Extensibility. Extend the classes already defined for QAST or write new ones. Or write new NQP ops.

" Simplicity. Yeah. About a dozen types of object written in a simple language (NQP).

" Program information. QAST annotations.

" Analysis information. QAST annotations.
```