Code Generator Development and Validation

Why SuperTest should be your first choice for compiler code generator development and validation!

By Marcel Beemster

SuperTest for code generator development and validation

The code generator is a critical part of the compiler that requires more specific testing than the rest of the compiler. SuperTest™ is the best test-suite to do this, as other test-suites are only designed to verify language compliance. Unlike other tools, SuperTest can be used almost from the start of new compiler development projects. Here is why:

SuperTest contains three specific strategies to put the code generator through its paces: using the handwritten tests aimed at code generation, using the depth suite and using the ABI-Tester.

In addition, the SuperTest suite is organized according to the language standard specification. This makes it easy to omit library testing from the initial test-runs. SuperTest also does not depend on a working library implementation and needs only a minimal interface to report the self-checking test results. SuperTest can be used very early in the code generation development process.

Code generator testing is an extremely relevant field of compiler testing today: most newly developed compilers rely on either the GCC or CLANG+LLVM compilers. For those compilers, few changes, if any, are needed in the front-end. Their front-ends are shared in many compiler implementations and are well tested (also by users of SuperTest).

Code generators, on the other hand, are target specific and are often aimed at specialized or application specific processors. They have a much smaller audience and rely on a smaller ecosystem to find and correct errors. This means that using high-quality validation tools for the code generator are required to achieve and maintain compiler quality. SuperTest is such a tool.

Mid-Level optimizations and the code generator

The code generator is that part of the compiler that must be modified when the compiler is retargeted to a different target processor. Three important tasks of the code generator are instruction selection, register allocation, and scheduling. In addition, there may be additional components in the code generator that perform other tasks. For example, 'lowering' (emulating) unsupported IR-level constructs and converting control flow into predicated execution are partly target specific.

The code generator is also called the 'back-end' because it is the final part of the compiler, after the front-end and the mid-level (mostly target independent) optimizations. For code generator specific tests to reach the code generator in their intended form, it is best to avoid the mid-level optimizations by turning optimizations

off. Otherwise, the mid-level optimizations may unintentionally transform the 'to be tested' programming construct into another form and fail to test the intended target instructions. Note that even without optimization options, some compilers do still optimize the code. If possible, it is best to turn this off as well.

To be sure, in later stages of code generator testing it is necessary to also test with optimizations turned on. The mid-level optimizations act like a pinball machine that may hand out strange curve balls to the code generator. This is good testing material, but it provides less control over exactly what is tested.

SuperTest hand written tests

The first line of defense against code generator errors is the huge collection of hand written SuperTest tests. SuperTest has been developed in close association with multiple generations of compiler development systems, including the CoSy® compiler development system. CoSy is a highly retargetable compiler development system that has been used for more than hundred different target architectures. Code generator development, and code generator validation, has always been of primary importance for CoSy.
For that reason, SuperTest contains a complete set of code generator specific tests. These were developed first by enumerating all combinations of constructs, operators and types relevant to the code generator, and then by using code coverage analysis for the code generator to find any gaps, both with and without optimizations turned on. In addition, errors found during code generator development (and other parts of the compiler) were also turned into SuperTest tests.

The Depth-Suite for target specific arithmetic

The second line of defense against code generator errors are the target arithmetic specific SuperTest Depth Suites. For many good reasons the C language does not specify the number of bits used for the arithmetic types. It only specifies that, for example for the 'int' type, at least 16 bits must be used. But that does not imply that C's arithmetic is loosely defined. To the contrary, the C specification does require that in an actual implementation, the sizes and precision used for arithmetic types ARE precisely defined. They are part of the 'implementation defined' data model.

It means that an implementation of C is free to choose the number of bits for arithmetic, but once that choice is made, it needs to be stated and adhered to. Then, based on this choice, the C specification states clearly how arithmetic must behave. For example, if an addition of two unsigned integers overflows, the results is wrapped around using the modulo of 1 greater than the maximum unsigned integer. In this case, the 'maximum unsigned integer' is an implementation defined parameter.

While the C definition of arithmetic is therefore quite precise, its dependence on implementation defined properties makes it hard for a general purpose test suite to verify its implementation exactly.

Yet SuperTest can do this. SuperTest includes more than thirty so-called 'depth suites'. SuperTest's depth suites are generated and contain millions of data model specific arithmetic tests for both integer and floating point types. These tests cover all combinations of arithmetic types, operators and values, in particular for the 'corner' cases that a general purpose test cannot begin to match.

And what if a particular data model is not one of the thirty included depth suites? Then we'll just create a new one on request.

The tests in the depth suites can also run with and without optimization. Without optimization they directly map to the arithmetic implementation in the code generator, which, as discussed, is a good thing. With optimization, they are also a great test for the mid-level, but target specific, constant folding transformations because constant folding also has to adhere to the rules of the target arithmetic.

The ABI-Tester for calling conventions and register allocation

SuperTest's third line of defense against code generator errors is the ABI-Tester (‘ABI’ for Application Binary Interface). The ABI-Tester is designed to give the target specific calling conventions implementation a good workout. The ABI-Tester is a generator that creates pairs of functions, with each element of the pair in its own file. One of the functions is the 'caller', it calls the other function called the 'callee'. The ABI-Tester generates random (but configurable) sequences of arguments that must be transferred from caller to callee. The functions are self-checking, so the callee verifies that it has received the right values, and the caller verifies the callee's return value.

The implementation of the calling conventions is closely linked to the register allocator and its optimizations. Register allocation is another target dependent part of the implementation and is complicated because an optimal register assignment is important for generated code performance. So, although that is not its main purpose, the ABI-Tester is also good at testing the register allocator.

Turning optimizations on does not affect the calling conventions themselves. But it does affect the 'register pressure' around function calls. With optimizations, many more registers are typically in use and the register allocator must make hard decisions about which values to keep in which registers. So here too, it makes sense to start with no optimization in the initial development of the compiler calling conventions and register allocator, and later raise the bar by turning optimizations on.

The utility of the ABI-Tester is not limited to testing the internal implementation of the calling conventions. The function pair can also be compiled by two different compilers

to verify that they both agree about the calling conventions implementation. Similarly, the pair can be compiled by two different versions of the same compiler in order to check that the calling conventions implementation has not changed between updates.

SuperTest: Your guard against code generation errors

All executable tests in SuperTest end up in the code generator, for the simple reason that the code generator is a necessary part of the compiler's functionality. This already provides substantial code generator test coverage.

In addition, SuperTest has three more lines of defense against code generator errors. SuperTest's hand-written tests contain a full complement of tests for all the operations that can be expected in the code generator. These tests were created and perfected over the course of developing multiple retargetable compiler systems.

The second line of defense is the depth suite, which is capable of testing the code generator's implementation of all arithmetic operations, types and their combinations. The depth suite verifies the target dependent 'corner cases' of arithmetic because it is generated for the specific implementation defined data model of the compiler.

The third line of defense is the ABI-Tester, which generates tests to verify the implementation of the calling conventions. In addition, it serves as a stress test for the implementation of the target dependent register allocator because that is intimately connected with the calling conventions implementation.

Finally, SuperTest is constructed so that it can even be used early in the development of the code generator.

That is why SuperTest should be your first choice for compiler code generator development and validation.