Adding support for C++ contracts to Clang

†Javier López-Gómez   *J. Daniel García
Computer Science and Engineering Department,Universisty Carlos III of Madrid

C++ contracts (P0542R5 TS)

P0542R5 is a proposal to support contract based programming in C++ which was recently accepted as part of the current ISO C++ working draft. It leverages a slightly modified C++11 attribute syntax to state assertions, preconditions or postconditions.

Assertions. The `[[assert: …]]` attribute appertain to a *NullStmt*. Hence, they may be located at any place where a *NullStmt* is valid, such as the body of a function. The associated predicate should hold at that specific point, much like the `assert()` macro.

Preconditions. They are used to state function expectations, i.e. what is expected from the user and are evaluated at function entry. The `[[expects: …]]` attribute is part of a function declarator; more formally, it appertains to the function type, although it is not part of it. The parsed expression may use anything currently in scope.

Postconditions. Also part of a function declarator, the attribute `[[ensures: …]]` specify what the function ensures after return, i.e. what the user can expect after the function returns. As such, they are evaluated before function exit. In addition to anything in scope, the parsed expression also has access to the return value.

Any of the previous attributes may optionally include an assertion level (referred to as $L_{contract\_attr}$ below). This may be one of:

axiom. Not evaluated at runtime, but may be used for other purposes, e.g. static analyzers or providing information to the optimizer).

default/audit. Intended to be used to indicate the relative computational cost of the checks.

A translation is carried out in a specific build level (off, default, audit). A check is enabled if:

$$L_{contract\_attr} \leq L_{build} \qquad (1)$$

Failed checks will by default cause the invocation of `std::terminate()`. Invocation of a user-defined handler is also possible. Optionally, `std::terminate()` may also be called after the user-defined handler returns (if continuation mode is off).

In a correct program, contracts have no observable effects beyond performance differences. Also, this is a convenient way to give additional information to the optimizer/third party libraries.

Listing 1: C++ contracts example

```
int f(int x)
    [[expects audit: x>0]]
    [[ensures r: r>0]] {
  /* … */
}
```

Opportunities for code optimization

Contracts provide additional information about the programmer's expectations which may be useful for the optimization passes.

If the continuation mode is "off", no special provision is required for checked contracts, as the violation handler is implicitly `[[noreturn]]`. Otherwise, only `axiom` contracts may be assumed (but not checked). This can be enabled by specifying the `-axiom-mode=on|off` compiler option.

Assumed contracts cause the emission of a call to `llvm::Intrinsic::assume` (the same used by `__builtin_assume()`).

Required changes to Clang

Overview

This required changes to some Clang components (shown in bold face in Figure 1):

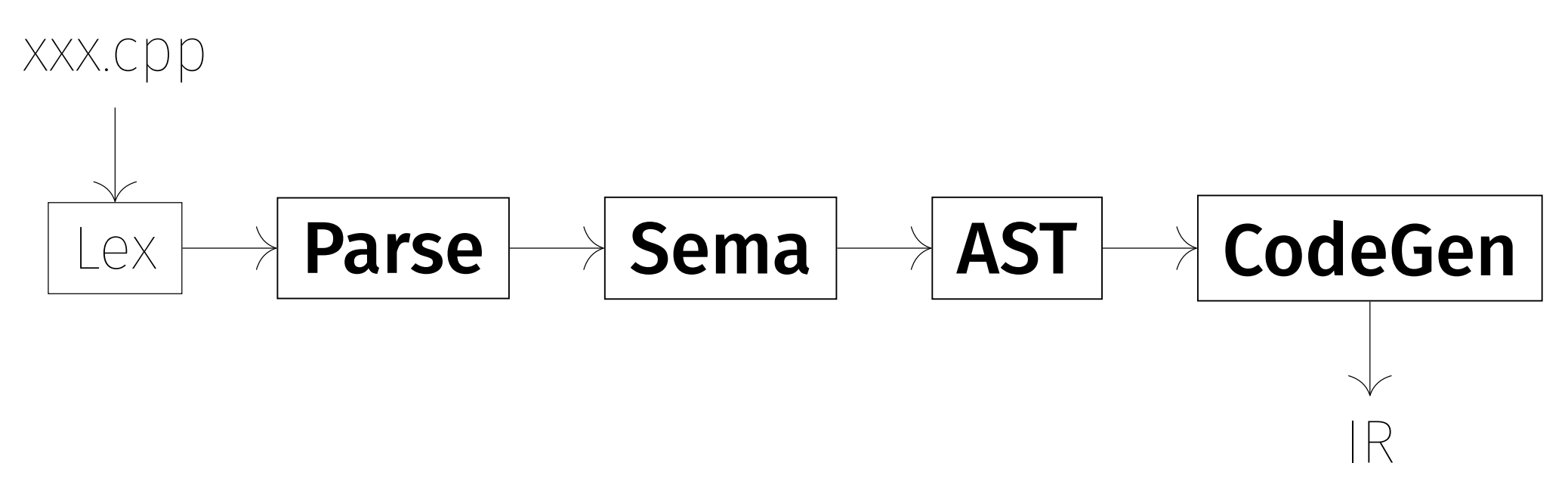

Figure 1: Patched Clang components

Parse. Updated to accommodate the grammar changes proposed in P0542R5 to support contract attributes, e.g. `[[expects audit: x==2]]`.

Sema. Most of the code lies here: injecting declarations, merging attributes, instantiation, etc.

AST. Small changes were done to the ASTContext and FunctionDecl classes to store additional information.

CodeGen. Code generation for the `[[assert]]` attribute and for checked functions.

The current implementation is publicly available at `https://github.com/arcosuc3m/clang-contracts/`.
Try it at `http://fragata.arcos.inf.uc3m.es/`.

Generating code for checked functions

Code generation for functions that have preconditions/post-conditions works as follows:

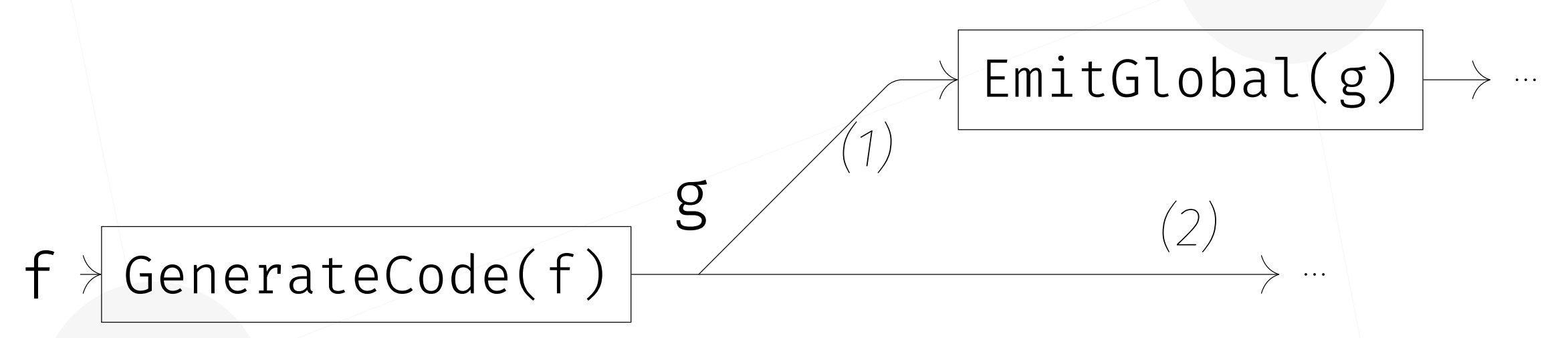

Figure 2: CodeGen for checked functions

At *(1)* the `f` FunctionDecl is copied; this copy (called `g` here) contains the original body of `f`, and has the `llvm::Attribute::AlwaysInline` attribute.
At *(2)*, the body of `f` is replaced by synthesized code that evaluates preconditions, makes a (inlined) call to `g`, and evaluates post-conditions.

```
int f(int x)
[[expects: x==2]]
{
  return x;
}
```

```
define i32 @_Z1fi ( i32
 returned %x)
local_unnamed_addr #0 {
entry:
  % cmp = icmp eq i32 %x, 2
  br i1 %cmp, label %if.end,
         label %if.then
if.then:
  tail call void
         @_ZSt9terminatev() #2
  unreachable
if.end:
  ret i32 2
}
```

Figure 3: C++ function and its LLVM IR

Evaluation

We replaced occurrences of the `__glibcxx_assert` macro found in the GNU implementation of the `std::basic_string` class by `[[assert: …]]` or `[[expects: …]]` attributes and compared the run-time overhead.

Here, we have included the following tests:

- Swap two characters of a `std::string` (10000 iterations) for varying string sizes (Figure 4).
- Find 3-char substring and replace each occurrence in a random string, for varying string sizes (Figure 5).

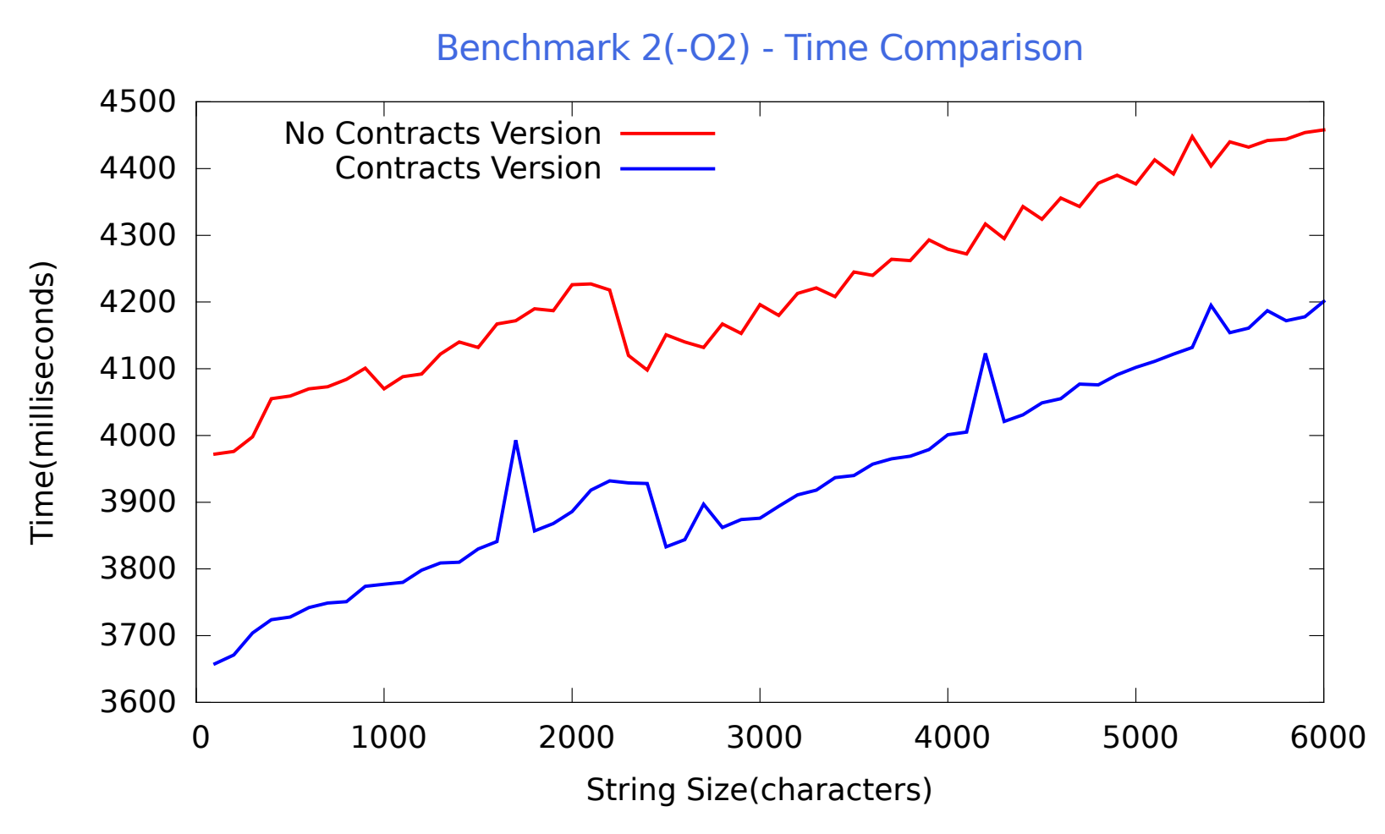

Figure 4: Character swap (-O2)

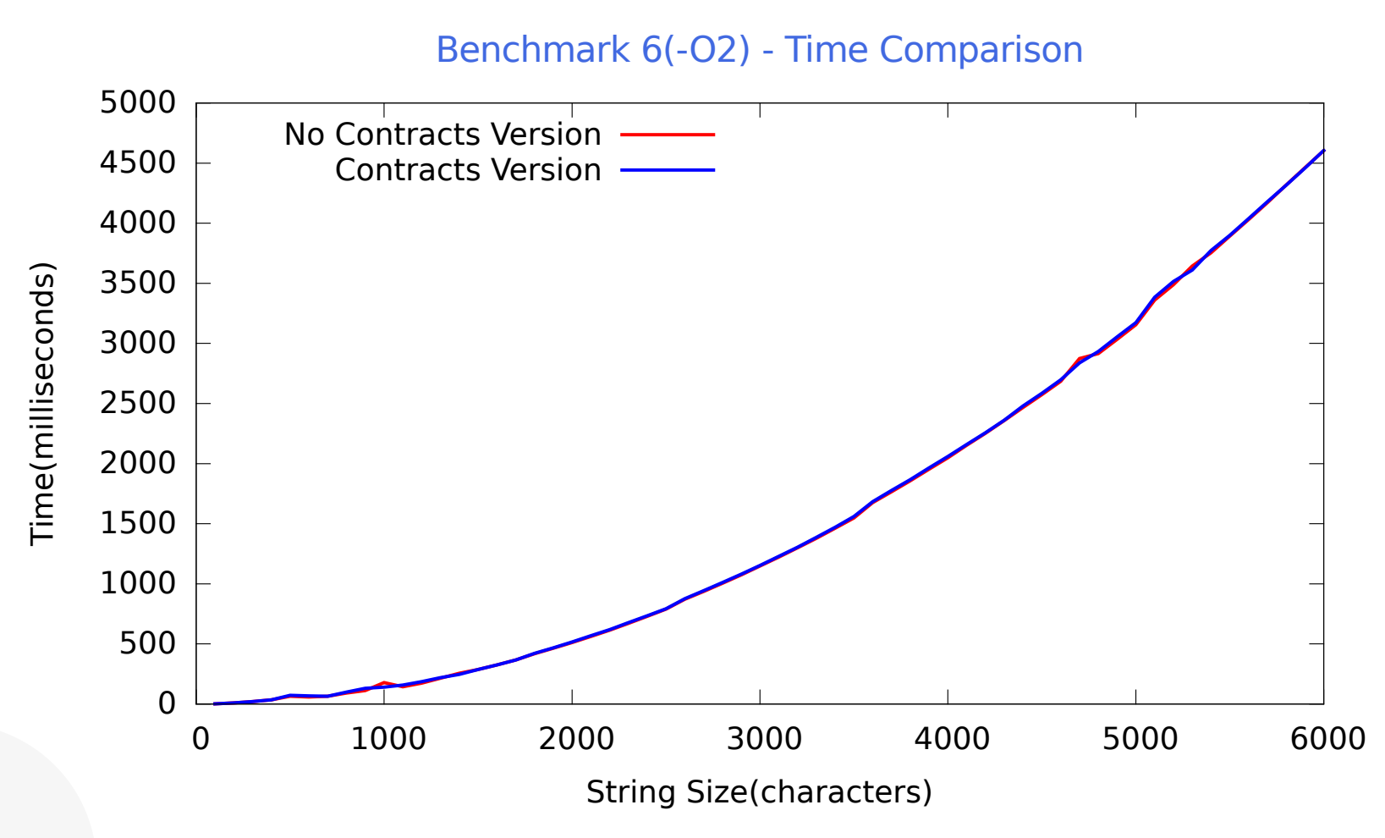

Figure 5: Replace substring (-O2)

Other interesting uses

On top of a P0542R5-enabled Clang, we built CSV, an extension to the ThreadSanitizer project that allows users to describe the semantics of lock-free data structures using C++ contracts.

The unmodified ThreadSanitizer reported false positives using a Boost lock-free SPSC queue (only 1 producer + 1 consumer). These false positives were filtered if CSV was in use.

To illustrate this point, we show an excerpt of the CSV annotations that were added to the `boost::lockfree:spsc_queue` class:

Listing 2: boost::lockfree:spsc_queue +CSV

```
// In header: <boost/lockfree/spsc_queue.hpp>
#include "csv.h"

template <typename T, typename... Options>
class [[csv::checked]] spsc_queue {
private:
  [[csv::event_sets(init_events, prod_events,
    cons_events,
       nts_events)]];
public:
  …
  bool push(T const &)
    [[expects audit: !init_events.empty()
        && init_events.happens_before(csv::
         current_event())]]
    [[expects audit: !prod_events.concurrent(
     csv::current_event())]]
    [[csv::add_current(prod_events)]]

  bool pop()
    [[expects audit: !init_events.empty()
        && init_events.happens_before(csv::
         current_event())]]
    [[expects audit: !cons_events.concurrent(
     csv::current_event())]]
    [[csv::add_current(cons_events)]]
  …
};
```

If built with a patched compiler + patched ThreadSanitizer, the aforementioned false positives are filtered. Additionally, if a rule is violated the user gets a descriptive trace.

CSV is open-source and is part of the `CSV-src` branch of the clang-contracts GitHub repository.

Conclusion

Support for contract-checking in C++...

- Enables to write more correct software by helping to detect more programming errors.
- Is a portable and standard way of providing information to the optimizer/third party libraries.

†jalopezg@inf.uc3m.es   *jdgarcia@inf.uc3m.es

https://github.com/arcosuc3m/clang-contracts/