Resumable Functions

Niklas Gustafsson
Microsoft Corp.

Resumable Functions

- std::future v2 provides:
  - Flexible and rich composition
  - Efficient support for immediately available results
  - Cancellation
  - Scheduling

- std::future v2 leaves unsolved:
  - Complexity of interacting with synchronous control flow, i.e. composing asynchronous code with loops, if-statements, goto, exception handling, etc.
  - Programmer unfamiliarity with “code inversion” pattern

Proposal

- Add two concepts to the language:
  - Resumable functions

    Multi-phase functions, identified at declaration:

```
future<int> f(stream str) resumable;
```

  - Resumption points

    Unary operator, only available in resumable functions, waiting for a promised value to become available:

```
int count = await str.read(512, buf);
```

Proposal

- To the caller, a resumable function behaves like any other function
  - Returns a container, which may initially be empty
  - The resumable function will eventually fill the container

- The compiler transforms the function definition to a non-blocking form
  - Allocating locals in heap-based storage, based on liveness analysis
  - Introducing a function state machine and state-based function prolog code
  - Transforming each resumption point locally
  - Transforming each return statement locally
  - No transformation of function declaration or calling convention

- Resumable functions may call other resumable functions as well as non-resumable functions

- Non-resumable functions may call resumable functions

Example I

```
future<int> f(shared_ptr<stream> str)
{
  shared_ptr<char> buf = ...;
  return str->read(512, buf)
    .then([](future<int> op) // lambda 1
          {
            return op.get() + 11;
          });
}

void g()
{
  shared_ptr<stream> s = ...;
  f(s).then([s](future<int> op) // lambda 2
            {
              s->close();
            });
}
```

Example I

```
future<int> f(stream str) resumable
{
  char buf[512];
  int count = await str.read(512, buf);
  return count + 11;
}

future<void> g() resumable
{
  stream s = ...;
  int pls11 = await f(s);
  s.close();
}
```

Example II

```
auto write = [&buf](future<int> size) -> future<bool>
  {
    return streamW.write(size.get(), buf)
      .then([](future<int> op){ return op.get() > 0; });
  };

auto flse = [](future<int> op){ return async::value(false);};

auto copy = do_while([&buf]() -> future<bool>
  {
    return streamR.read(512, buf)
      .choice([](future<int> op){ return op.get() > 0; }, write, flse);
  });
```

Example II

```
int cnt = 0;
do
{
  cnt = await streamR.read(512, buf);
  if ( cnt == 0 ) break;
  cnt = await streamW.write(cnt, buf);
} while (cnt > 0);
```

Feasibility

- We don't have a C++ prototype, but…
  - C#/VB is shipping a very similar implementation in the next Visual Studio
    - C# iterators, which has been shipping for many years, are based on similar local transformations
  - F# has supported similar features since v1.0
  - Python also supports similar functionality
  - The code transformations are simple and local