Date: April 22, 2012
Reply to: Niklas Gustafsson
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052-6399
Email: niklas.gustafsson@microsoft.com

Resumable Functions

This is a revised version of the document presented in February 2012, in Kona, HI. It has been edited for clarity. The previous version described a set of compiler transformations I their general form, which lead to unnecessary confusion; this version describes the transformation exclusively in terms of *future<T>* and *shared_future<T>* and leave the generalization to the reader.

As in the previous version, this proposal is related to the proposal for improvements to the std::future library. The reader is advised to read both as a unit and to consider how the two build on each other for synergy. Reading the proposal on asynchronous operations before this one is recommended.

1. The Problem

The direct motivation for introducing resumable functions is the increasing importance of efficiently handling I/O in its various forms and the complexities programmers are faced with using existing language features and existing libraries.

The motivation for a standard representation of asynchronous operations is outlined in the accompanying proposal and won't be repeated here. The need for language support for resumable functions arises from the inherent limitations of the library-based solution described in that proposal.

Taking a purely library-based approach to composition of asynchronous operations means sacrificing usability and versatility: the development of an asynchronous algorithm usually starts with a synchronous, imperative expression of it, which is then manually translated into an asynchronous equivalent. This process is quite complex, akin to the reformulation of an imperative algorithm in a pure functional language, and the resulting code may be difficult to read.

As discussed at the end of the accompanying proposal, a library-based approach leads to object lifetime management complexities and thus a different way of designing the objects that are to be used by and with asynchronous operations.

A pure library-based approach further complicates the ability to support the model with tooling: asynchronous algorithms usually result in an inversion of control, such that activation frames appear "upside down" in a debugger.

Consider this example, using the modified version of future<T> introduced by the accompanying proposal:

```
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();
          });
}
```

When g() is activated, it creates its stream object and passes it to f, which calls the read() function, attaches a continuation (lambda 1) to its result, and then returns the result of the then() member function call. After the call to f() returns, g attaches a continuation (lambda 2) to the result, after which it returns to its caller.

When the read operation finishes, lambda 1 will be invoked from some context and its logic executed, resulting in the operation returned from f() completing. This, in turn, results in lambda 2 being invoked and its logic executed. If you were to set a breakpoint in either of the lambdas, you would get very little context or information on how you got there, and a debugger would be hard-pressed to make up for the lack of information.

To make matters worse, the above code does not consider that the futures returned by read() and f() may already be completed, making the attachment of a continuation lambda unnecessary and expensive. To squeeze out all the performance possible, the code will wind up being quite complex.

Contrast this with how the same algorithm, just as efficient and asynchronous, would look when relying on resumable functions:

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

future<void> g() resumable
```

```
{
    stream s = ...;
    int pls11 = await f(s);
    s.close();
}
```

Not only is this simpler, it is more or less identical to a synchronous formulation of the same algorithm.

The library-based approach gets even more complicated when our example includes control-flow such as conditional evaluation and/or loops. The language-based approach allows control-flow to remain identical to the synchronous formulation, including the use of try-catch blocks and non-reducible constructs such as goto and break.

The following example illustrates this.

While iterative composition is not covered in the accompanying proposal, it is mentioned that iteration can be created in terms of the defined primitives. With a do_while() construct (a few lines of templates-based code), we get this logic for a loop that reads from one file and writes to another:

```
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);
  });
```

With resumable functions, the same code snippet would be:

```
int cnt = 0;

do
{
    cnt = await streamR.read(512, buf);

    if ( cnt == 0 ) break;

    cnt = await streamW.write(cnt, buf);

} while (cnt > 0);
```

It is not necessarily a lot shorter, but undoubtedly easier to comprehend; it's more or less identical to a synchronous formulation of the same algorithm. Further, no special attention needs to be paid to object lifetimes of variables within the function.

Resumable functions are motivated by the need to adequately address asynchronous operations, but are not technically tied to the proposal for a standard representation of such operations. The proposed compiler transformations can be generalized to cover other types than *future<T>* and *shared_future<T>*.

For the sake of clarity, based on feedback from Kona, this generalization will not be spoken of again.

2. The Proposal

2.1 Terminology

A resumable function is a function that is capable of split-phase execution, meaning that the function may be observed to return from an invocation without producing its final logical result or all of its side-effects. This act is defined as the function pausing its execution. The result returned from a function when it pauses is a placeholder for the logical result, for example a *future<T>* representing a function that eventually returns a value of type T.

After pausing, a resumable function may be resumed by the scheduling logic of the runtime and will eventually complete its logic, at which point it executes a return statement (explicit or implicit) and sets the result value in the placeholder future. Executing a return statement is herein referred to as logically returning from the function.

Within the function, there are zero or more resumption points. A resumable function may pause when it reaches a resumption point. Given control-flow, it may or may not be the case that a resumable function actually reaches a resumption point before logically returning.

There are two distinct compiler transformations in this proposal; they are related but should be not to be confused:

The first transforms the implementation of the function itself, mostly hiding the fact that it is a resumable function from the caller, while fundamentally altering its internal structure and strategy toward allocation of storage for local variables.

The second transformation regards the added unary operator ‘await.’

2.2 Declaration and Definition

Resumable functions are identified by placing the identifier ‘resumable’ after the parameter list. In the case of a virtual function, all instances of the virtual function need to have the same designation, either resumable or not.

Whether or not a function is resumable is ignored when performing function overload resolution, but it is considered part of its type signature. A pointer to a resumable function will have to be converted in order to be treated as a pointer to a non-resumable function.

Resumable functions cannot use a variable number of arguments. For situations where varargs are necessary, the argument unwrapping may be placed in a function that calls a resumable function after doing the unwrapping of arguments.

The return type S of a resumable function "S f() resumable" can be *future<T>* or *shared_future<T>,* where T may be 'void.'

The compiler transformation depends on the existence of a default constructor on *promise<T>*, the existence of set_value() and set_exception(), and the ability to create a *future<T>* from a *promise<T>* and a *shared_future<T>* from a *future<T>*.

A resumable function logically returns[1] (produces its final value) when executing a return statement or, in the case of *future<void>* or *shared_future<void>* as the return type, it reaches its end without executing a return statement.

For example:

```
future<int> abs(future<int> i_op) resumable
{
    int i = await i_op;
    return (i < 0) ? -i : i;
}
```

2.3 Resumption Points

Within a resumable function, its resumption points are uniquely identified by the presence of the unary operator 'await', which is treated as a keyword or reserved identifier within resumable functions.

```
await expr
```

The choice of the word "await" comes from the notion that the operand may be empty and that the code will wait for it to be filled.

Note:

The await operator takes an operand of type future<S> or shared_future<S> and produces a value of type S. The S used in such an expression within the body of a resumable function returning future<T> does not have to have any relation to T, they are entirely independent. Further, there need not be any relation between the types S, S', S'', etc. used in distinct await expression within the resumable function.

[1] The implementation is expected to allocate a promise<T> to represent the production side of the result. In the return statement (or when the end is reached), this promise's set_value() function is called.

In order to be used with the await operator, S must have a public copy constructor (custom or compiler-provided) or be a primitive type, or 'void.' An assignment operator must be available for S unless S is 'void.'

If S is 'void,' the expression must be the expression term of an expression statement.

3. Interactions and Implementability

3.1 Interactions

(Contextual) Keywords

The proposal uses two special identifiers as contextual keywords to declare and control resumable functions. These should cause no conflict with existing, working, code.

In the case of 'resumable', it appears in a place where it is not currently allowed and should therefore not cause any ambiguity. Introducing the use of resumable as an identifier with a special meaning only when it appears in that position is therefore not a breaking change.

In the case of 'await,' it is only reserved within the body of a resumable function; since there are no existing resumable functions, its introduction is not a breaking change.

A possible conflict, but still not a breaking change, is that the identifiers may be in use by existing libraries. In the case of 'resumable,' the context should remove the possibility of conflict, but 'await' is more difficult. When used with a parenthesized operand expression, it will be indistinguishable from a call to a function 'await' with one argument.

A second possible non-breaking conflict is if there is a macro of the name 'await,' in which preprocessing will create problems.

A quick search of the header files for the Microsoft implementation of the standard C++ and C libraries, the Windows 7 SDK, as well as a subset of the Boost library header files show that there are no such conflicts lurking within those common and important source bases.

Declarations vs. Definition

I considered making resumable a property of the function definition rather than the function declaration, but rejected it for the following reasons:

i. A function being resumable places constraints on the signature of the function by disallowing ellipsis arguments and void return types. While this can be flagged as an error when the definition is processed, it seems more reasonable to enforce such constraints as early as

possible.

ii. Given the inherent concurrency of a resumable function that pauses vis-à-vis its caller, passing a reference or pointer to a local variable into a resumable function may be dangerous. This risk can be flagged as a warning by a compiler or code analysis tool only if the resumable property is part of its type signature.

iii. By having the resumable property on the declaration, an implementation can, if it chooses, introduce a new calling convention for use with resumable functions and apply it at the call site.

Overload Resolution

By ignoring the resumable property of a function when performing overload resolution, the interaction with overloading should be minimal. From the perspective of a caller, there is nothing special about a resumable function when considering overloads.

Expression Evaluation Order / Operator Precedence

This proposal introduces a new unary operator, only valid within resumable functions. The most significant interaction with other language elements is the precedence of the new operator, a detail which is left out of the proposal for consideration at a later time.

3.2 Implementability

While there is currently no prototype implementation of the proposed solution, the existence of almost identical solutions in other languages, including Python, C#, VB, and F#, gives me great confidence in its feasibility and utility. The implementation is also similar to the implementation of methods in C# that use the 'yield return' statement, a construct that has been available in C# for many years and is well vetted.

I have "hand-translated" resumable functions and await expressions into a form that complies with the proposal's semantics. It is described in the following sections as a possible implementation. Whereas hand-translating the function produces correct C++ code, a compiler would obviously choose a lower-level representation. For the purpose of pedagogy and brevity, the following sections express the translation as C++.

3.2.1 Function Declaration

The declaration of a resumable method is implemented by introducing an additional function declaration with an appropriately mangled name and a structure to hold local function state across resumption points.

```
future<int> f(future<double> g) resumable;
```

Produces, in “hand-translated” C++ terms:

```
// A declaration of the function as seen from the outside.
// The definition of this function will bridge the different
// runtime models of the caller and the resumable function
future<int> f(future<double> g);

// A structure to hold locals. A compiler would not bother creating
// a name for it.
struct _frame_f;

// A mangled-name function that will hold the transformed body of
// the resumable function.
void _res_f(std::shared_ptr<_frame_f> frame);
```

If ‘f’ is declared in class scope, the additional declarations are added as private declarations of the class.

Depending on the implementation, the declarations may or may not actually be manifested in the name space. For example, the second function may simply be a second entry point of “f,” and “_frame_f” may simply be a reference-counted byte array.

3.2.2 Function Definition

The definition of a resumable function results in the definition of the locals frame structure and the added function, into which the body of the resumable method is moved before being transformed. The resumable function itself is more or less mechanically changed to allocate an instance of the frame structure, copy or move the parameters, and then call the new function.

It’s worth once more pointing out that the frame structure is a by-product of our attempt to represent the transformations using valid C++ code. A “real” implementation would allocate a suitably large byte array and use that for storage of local variables and parameters. It would also run constructors and destructors at the correct point in the function, something that our source-code implementation cannot.

The definition:

```
future<int> f(future<double> g) resumable { return ceil(await g); }
```

results in:

```
struct _frame_f
{
    int _state;
    future<int> _resultF;
    promise<int> _resultP;
    _frame_f(double g) : g(g), _state(0)
    {
        _resultF = _resultP.get_future();
    }
    double g;
};
```

```
future<int> f(double g)
{
    auto frame = std::make_shared<_frame_f>(g);

    _res_f(frame);

    return frame->_resultF;
}

void _res_f(const std::shared_ptr<_frame_f> &frame)
{
    return ceil(await g);
}
```

Note that the body of the _res_f() function takes some artistic license, as it represents a transitional state of the original body. It still needs to be transformed into its final form, as described in the next section.

3.2.3 Function Body

There are four main transformations that are necessary, not necessarily performed in the order listed: a) space for local variables needs to be added to the frame structure definition, b) the function prolog needs to branch to the last resumption point, c) await expressions need to be hoisted and then transformed into pause/resumption logic, and d) return statements need to be transformed to modify the _result field of the frame.

3.2.4 Allocating Storage

All variables (and temporaries) with lifetimes that statically span one or more resumption points need to be provided space in the heap-allocated structure. In the hand-translated version, their lifetimes are extended to span the entire function execution, but a real, low-level implementation must treat the local variable storage in the frame as just storage and not alter the object lifetimes in any way.

The heap-allocated frame is reference-counted so that it can be automatically deleted when there are no longer any references to it. In this source-code implementation, we're using std::shared_ptr<T> for reference counting. Something more tailored may be used by a "real" implementation.

An implementation that cannot easily perform the necessary lifetime analysis before allocating space in the frame should treat all local variables as if their lifetimes span a resumption point. Doing so will increase the size of the heap-allocated frame and decrease the stack-allocated frame.

3.2.5 Function Prolog

The "_state" field of the frame contains an integer defining the current state of the function. A function that has not yet been paused always has _state == 0. With the exception of the initial state, there is a

one-to-one correspondence between state identities and await operators. The order in which state identities is assigned has no significance, as long as each identity uniquely identifies a resumption point.

Each state is associated with one label (branch target), and at the prolog of the function is placed the equivalent of a switch-statement:

```
    void _res_f(std::shared_ptr<_frame_f> frame)
    {
        switch(frame->_state)
        {
        case 1: goto L1;
        case 2: goto L2;
        case 3: goto L3;
        case 4: goto L4;
        }
    }
```

In the hand-coded version, special care has to be taken when a resumption point is located within a try-block; an extra branch is required for each nesting try block: first, the code branches to just before the try-block, we allow the code to enter the block normally and then branch again:

```
    void _res_f(std::shared_ptr<_frame_f> frame)
    {
        switch(frame->_state)
        {
        case 1: goto L1_1;
        case 2: goto L2;
        case 3: goto L3;
        case 4: goto L4;
        }

L1_1:
        try
        {
            switch(frame->_state)
            {
            case 1: goto L1;
            }

L1:
            ...
        }
    }
```

Depending on the implementation of try-blocks, such a chaining of branches may not be necessary in a more realistic low-level expression of the transformation.

3.2.6 Hoisting ‘await’ Expressions

Before transformation, each resumption point needs to be in one of these two forms:

```
        x = await expr
```

```
await expr
```

In other words, embedded await operators need to be hoisted and assigned to temporaries, or simply hoisted in the case of void being the result type. The operand 'expr' also needs to be evaluated into a temporary, as it will be used multiple times in the implementation, before and after the resumption point.

3.2.6 Implementing 'await'

In our implementation, "t = await g" is transformed thus:

```
        if ( !frame->g.is_done() )
        {
            frame->_state = 1;
            frame->g.then(
              [=](future<int> op)
              {
                  __res_f(frame);
              });
            return;
        }

L1:
        t = frame->g.get();
```

In the case of 'await g' being used as the expression of an expression statement, i.e. the value is thrown away, the compiler will still emit the call to 'get()' after the resumption. Calling get() even when the result is not used gives the runtime a chance to raise any propagated exceptions that may otherwise go unobserved.

3.2.7 Transforming 'return' Statements

Return statements are simply transformed into calls to set the value contained in the _result objects, or overwrite it with a new object:

```
        // return ceil(await g);

        if ( frame->_state == 0 )
            frame->_resultF = create_value<int>(ceil(t));
        else
            frame->_resultP.set_value(ceil(t));
```

The test for _state == 0 is done to establish whether the function has ever been paused or not. If it has not, it means that the caller will not have been passed back the result instance and it is therefore not too late to replace it. In the case of async operations, as outlined in the accompanying proposal, this enables an important optimization by allowing the resumable function to return a prompt operation.