Prototyping Async Iteration

https://tc39.github.io/proposal-async-iteration/

1. Async Iterator protocol

Similar to the ES6 Iterator protocol, Async Iteration introduces a new Symbol noted as *@@asyncIterator*. *GetIterator()* now selects whether to create an async or sync iterator depending on a flag passed to the spec algorithm. In the current proposal, it is always statically known whether it’s necessary to create an async or sync iterator, which is the good news.

Currently, the only user of GetIterator(iterable, “async”) is the for-await loop syntax, defined in https://tc39.github.io/proposal-async-iteration/#sec-for-in-and-for-of-statements.

Example implementation:

```
void BytecodeGenerator::BuildGetIterator(Expression* iterable, GetIteratorHint hint =
GetIteratorHint::kNormal) {
  RegisterList args = NewRegisterList(1);
  Register method = NewRegister();

  VisitForAccumulatorValue(iterable);
  builder()->StoreAccumulatorInRegister(args[0]);

  If (hint == GetIteratorHint::kAsync) {
    // Set method to GetMethod(obj, @@asyncIterator)
    builder()->LoadNamedProperty(args[0], async_iterator_symbol(),
                                 NewFeedbackSlot(LOAD_IC));

    BytecodeLabel async_from_sync, done;
    builder()->JumpIfUndefined(&async_from_sync);

    builder()->StoreAccumulatorInRegister(method);
    builder()->Call(method, args, NewFeedbackSlot(CALL_IC), Call::NAMED_PROPERTY_CALL);

    // If Type(iterator) is not Object, throw a TypeError exception.
    builder()->JumpIfIsJSReceiver(&done);
    builder()->CallRuntime(Runtime::kThrowSymbolAsyncIteratorInvalid);


    builder()->Bind(&async_from_sync);
    // If method is undefined,
    //     Let syncMethod be GetMethod(obj, @@iterator)
    builder()->LoadNamedProperty(args[0], iterator_symbol(), NewFeedbackSlot(LOAD_IC));
    builder()->StoreAccumulatorInRegister(method);

    // Let syncIterator be Call(syncMethod, obj)
    builder()->Call(method, args, NewFeedbackSlot(CALL_IC), Call::NAMED_PROPERTY_CALL);
    builder()->StoreAccumulatorInRegister(sync_iterator);

    // %_CreateAsyncFromSyncIterator(syncIterator)
    builder()->CallRuntime(Runtime::kInlineCreateAsyncFromSyncIterator);

    builder()->Bind(&done);
    return;
  }

  // regular sync iterator stuff...
}
```

The AsyncFromSync iterator type is a special implementation of AsyncIterator, which simply wraps a sync iterator returned from iterable.@@*iterator()*, and wraps iterator results as Promises. It's implementation is similarly straight forward, building on top of Sathya's Promise refactoring, and the ability for TFJ builtins to catch exceptions. These methods will always turn caught exceptions into rejected promises, which fits nicely into the current TFJ catch prediction approach.

2. Async Generators

Async Generators can be thought of as a combination between ES6 Generators, and ES2017 Async Functions, supporting suspension through both AwaitExpressions and YieldExpressions.

In addition to this complexity above, Async Generators are specified to have a list of "request" objects (*AsyncGeneratorRequest*), which are processed in order. The *AsyncGeneratorResolve()* and *AsyncGeneratorReject()* operations essentially exhaust the list of requests, apparently within the same microtask. *(i)*

The AsyncGeneratorRequest could be implemented as either a FixedArray (similar to Promise handler FixedArrays), or as a singly linked list. Though it is possible for there to be multiple requests at a single time, a single for-await-of loop will only queue up a single request per iteration step, so it's likely that there will only be at most one request per turn.

General design:

Class JSAsyncGeneratorObject extends JSGeneratorObject, adding a builtin property for the list of requests (either a single pointer to hold a FixedArray, or a pointer to the head of the singly linked list (which is replaced with *AsyncGeneratorRequest.next* when removed from the set), solving *i*. Adding a pointer to the tail end of the list would speed up what I believe is an uncommon case, but most likely not by a significant amount.

%AsyncGeneratorPrototype%.next, .throw and .resume: TF_BUILTINs with JS linkage, which each perform *AsyncGeneratorEnqueue()*, which is most likely implemented as a code-stub tail called from the caller. AsyncGeneratorEnqueue creates an AsyncGeneratorRequest object, adds it to the end of the list, and tests the state of the generator. If the generator is executing currently, it aborts, because during execution, the list of requests will be exhausted anyways. Otherwise, it performs *AsyncGeneratorResumeNext*, which loops through the set of requests in FIFO order, and triggers the generator resume method with information from the current request (including the resume mode to use, and the value to pass to the generator).

When the generator is suspended via a yield expression, new bytecode is generated to perform *AsyncGeneratorResolve(generator, value, done)* rather than simply returning an iterator result. However, when the generator is suspended via an await expression, for compatibility with the current async functions implementation, it behaves as though the generator is synchronous (as it currently does in v8's Async Functions implementation). **Note: This situation is awkward when combined with the function.sent metaproperty, as it seems strange for an awaited value to become mapped to *function.sent*.**