Storing the home object in Context instead of JSFunction

marja@ - January 2021 - Attention: visible externally!

Why?

- Loading it from the context is faster than the indirection via JSFunction.
- Saves memory. E.g., for methods, we have a JSFunction per method, whereas they all belong to the same class Context.
- Currently, the optimized code generated by --super-ic is not as fast as it could be, because it doesn't realize the home object in a JSFunction is a constant. We could fix it there, but it makes more sense to do this project now and then fix the potential constantness.
    - See this doc for a detailed analysis of the optimized code both before and after this work.

Below are some examples which demonstrate the complexities in super property access.

The simple cases

```
class A {};
A.prototype.x = 'right';

class B extends A {
  m() { return super.x; }
};
```

Previously, the home object (B.prototype) was stored in the JSFunction for m. Now it'll be stored in the Context for B.

Super property references inside static methods work differently:

```
class A {};
A.x = 'right';

class B extends A {
  static m() { return super.x; }
};
```

For static methods, the home object is B, and the home object's __proto__ is A.

To this end, we store two home objects in the Context: the “normal” home object and the “static home object” (home object for static methods).

For simplicity, we always store both (if at least one is needed).

Super in static functions

Currently, when generating the bytecode for a function, we don’t know whether it’s a static function of some kind (static method, static generator, static async generator etc). This information needs to be added, so that the bytecode generator can direct the super property access to the static home object. Previously, this was dealt with by storing the correct home object in the JSFunction itself.

To this end, I extended `FunctionKind` to keep track of various static function kinds.

Super in object literals

```
(function TestSuperInObjectLiteralMethod() {
  let my_proto = {
    __proto__ : {'x': 'right' },
    m() { return super.x; }
  };
  let o = {__proto__: my_proto};
  assertEquals('right', o.m());
})();
```

Previously, the home object (my_proto) was stored in the JSFunction for m. Now we need to create a Context for the object literal and store the home object there.

When parsing the object literal, we need to create a Scope (which the Context then corresponds to). If the Context isn’t needed (no super references), we destroy the Scope and don’t insert it into the scope tree.

This scope is only used when parsing **methods** inside an object literal. (Getters and setters behave similarly to methods.)

In particular, the object literal scope it’s not used for property values:

```
(function TestSuperInObjectLiteralProperty() {
  class OuterBase {};
  OuterBase.prototype.x = 'right';
  class Outer extends OuterBase {
    m2() {
      let my_proto = {
        __proto__ : {'x': 'wrong' },
        m: () => super.x,
      };
      let o = {__proto__: my_proto};
      return o.m();
    }
  }
  assertEquals('right', (new Outer()).m2());
})();
```

Here the super reference inside the arrow function behaves like any super reference inside m2() (but not inside any object literal methods).

Since the object literal scope is only used for methods, the Scopes for functions don't match the syntax 1-to-1:

```
let object = { // object literal 1 starts
  __proto__: { // object literal 2 starts
    method1() {}
  }, // object literal 2 ends
  method2() {
    return super.method1();
  }
}; // object literal 1 ends
```

Now method1 is inside the object literal 1 syntactically, but scope-wise it's not:

```
Global scope:
global { // (0x5578eca6e5c8) (0, 205)
  // will be compiled
  // 1 stack slots
  // 3 heap slots
  // temporary vars:
  TEMPORARY .result;  // (0x5578eca6f248) local[0]
  // local vars:
  LET object;  // (0x5578eca6e800) context[2]

  function method1 () { // (0x5578eca6eb38) (95, 100)
    // lazily parsed
    // 2 heap slots
  }

  block { // (0x5578eca6e830) (13, 178)
```

```
    // 3 heap slots
    // local vars:
    CONST .home_object;  // (0x5578eca6f050) context[2], forced context
allocation, never assigned
    // home object var    CONST .home_object;  // (0x5578eca6f050) context[2],
forced context allocation, never assigned

    function method2 () { // (0x5578eca6ee68) (140, 176)
      // lazily parsed
      // 2 heap slots
    }
  }
}
```

This happens only for methods which don’t access the home object.

Super in property initializers

Super in non-static property initializers

```
(function TestSuperInsidePropertyInitializer() {
  class OuterBase {}
  OuterBase.prototype.prop = 'wrong';
  class Outer extends OuterBase {
    m() {
      class A { }
      A.prototype.prop = 'right';

      class B extends A {
        x = () => { return super.prop; };
      }

      B.prototype.prop = 'wrong';
      return (new B()).x();
    }
  }
  Outer.prototype.prop = 'wrong';

  assertEquals('right', (new Outer()).m());
})();
```

The property initializers are inside a “property initializer function”. Previously, that function had its own home object. Now we need to make sure they access the home object in the Context.

Super in static property initializers

```
(function TestSuperInsideStaticPropertyInitializer() {
  class OuterBase {}
  OuterBase.prototype.prop = 'wrong';
  OuterBase.prop = 'wrong';

  class Outer extends OuterBase {
    m() {
      class A { }
      A.prop = 'right';
      A.prototype.prop = 'wrong';
      class B extends A {
        static x = super.prop;
      }
      B.prop = 'wrong';
      B.prototype.prop = 'wrong';
      return B.x;
    }
  }
  Outer.prototype.prop = 'wrong';
  Outer.prop = 'wrong';

  assertEquals('right', (new Outer).m());
})();
```

The static property initializers are inside a “static property initializer function”. Previously, that function had its own home object. Now we need to make sure they access the “static home object” in the Context - the same way a static method would.

Super in the “extends” clause

Super property access inside the “extends” clause binds outside the class whose “extends” clause we’re parsing. (See ClassScope::HeritageParsingScope, Scope::IsParsingHeritage, Scope::private_name_lookup_skips_outer_class_.)

Example with a Proxy:

```
(function TestSuperInsideExtends() {
  class C extends class {
```

```
                  static a = 'right';
                } {
    static m = class D extends new Proxy(function f() {},
                                          {get:(t, k) => {
                                             if (k == "prototype") {
                                               return
Function.prototype;
                                             }
                                             return super.a;
                                           }
                                          }) {}
  };
  assertEquals('right', C.m.a);
})();
```

Same example but without a Proxy:

```
(function TestSuperInsideExtends2() {
  function f(x) {
    function A() { }
    A.x = x;
    return A;
  }

  class B {};
  B.a = 'right';

  // How to write "super" inside the extends clause? The "class extends value"
  // needs to be a constructor.
  class C extends B {
    static m = class D extends f({m2: () => { return super.a;}}) { }
  }

  // C.m is a class. Its "parent class" is a function (returned by f). C.m.x
  // binds to the parent's x, which is whatever we passed as a param to f.
  // In this case, it's an object which has a property m2.

  // Since m2 is an arrow function, and not a method, "super" inside it
  // doesn't bind to the object where m2 is defined, but outside.
  assertEquals('right', C.m.x.m2());
})();
```

Super and the preparse data

Example:

```
class A {}

class B extends A {
  m() {
    let my_proto = { b() { return super.y; }};
  }
}

(new B).m();
```

First, m is lazy-parsed. When m is called, it's eager-parsed and b is skipped.

We need to keep track of the super reference inside b, so that we correctly figure out that the object literal scope needs a Context when eager-parsing m.

We modify the existing implementation to be more imprecise (but correct enough for our purposes): we track that b's home object scope (the object literal scope) needs a home object and save that information in the preparse data for b. This replaces the UsesSuper bit with MaybeUsesSuper bit. The latter is true also when the skipped function contains an eval.

When skipping a function, we use its MaybeUsesSuper bit to set Scope::needs_home_object_ in the home object scope of the function.

List of changes in the CL

- Various symbol machinery can now be removed, as those parts were only used for the home_object_symbol (which is now gone).
- ObjectLiteral now stores Variable* home_object
    - Scope can be retrieved from it and used for its methods
- ClassLiteral stores Variable* home_object, static_home_object
- SuperPropertyReference stores a VariableProxy for the home object, found based on the name (the name is different for static and non-static methods)
- Scope::GetHomeObjectScope implements finding the Scope where super property references associate to (see the part about "extend" clauses)
- Functions no longer need home objects
    - No separate FunctionModes based on "needs vs doesn't need"
    - No separate maps for JSFunctions which need vs don't